UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

TRABAJO FIN DE GRADO

Ingeniería Electrónica Industrial y Automática

PROYECTO DE ASISTENCIA AL CONDUCTOR BASADO EN EL DISEÑO DE UN SISTEMA DE AVISO Y DETECCIÓN DE ADELANTAMIENTOS

AUTOR: Mario Fernández Rodríguez

TUTOR: Pablo Marín Plaza

SEPTIEMBRE 2014

Título: Proyecto de asistencia al conductor basado en el diseño de un sistema de aviso y detección de adelantamientos.

Autor: Mario Fernández Rodríguez

Tutor: Pablo Marín Plaza

TRIBUNAL

Presidente: Luis Hernández Corporales

Secretario: Jonathan Crespo Herrero

Vocal: Fernando de Terán Vergara

Realizado el acto de defensa y lectura del trabajo Fin de Grado el dia 2 de Octubre de 2014 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de _____

PRESIDENTE

SECRETARIO VOCAL

Agradecimientos

En primer lugar a mi tutor, Pablo Marín, por haberme permitido participar en el proyecto y por la ayuda prestada para ir solucionando los problemas del mismo; y a aquellos profesores que se han preocupado por formarme como ingeniero y como persona.

En segundo lugar a mis compañeros de universidad, tras miles de horas de estudio juntos, y al resto de mis amigos, sin su comprensión y apoyo no estaría aquí.

Y en tercer lugar pero más importante a mi familia: a mi hermana Sonia por su preocupación continua e interés en ayudarme a lo largo del proyecto y de la carrera. A mi madre Encarna, por ser el soporte que me ha mantenido en pie en los peores momentos de estos años. Y a mi padre José Joaquín, porque si alguien me ha guiado hacia esta rama de la Ingeniería ha sido él.

Índice general

Índice de figuras

Lista de anexos

Resumen

En la presente memoria se explica el desarrollo de un algoritmo que permita detectar adelantamientos a un turismo que disponga de una cámara embarcada a partir de sus imágenes y su procesado.

La aplicación descrita será capaz de advertir al conductor cuando se produzcan dichos adelantamientos así como de proporcionar información de la posición del vehículo que podrá ser utilizada posteriormente para posibles expansiones del proyecto.

Palabras clave: Ubuntu, Robot Operating System, OpenCV, Flujo óptico, Detección de esquinas, Detección de adelantamientos.

Abstract

Every day, drivers die in road collisions. The majority of road crashes are caused by human error. Research has shown that driver error accounts for over 80% of all fatal and injury crashes.

Most of these errors are related to overtakings.

The aim of this project is to develop an algorithm to detect an overtaking vehicle using images obtained from a camera and futher processing.

Keywords: Ubuntu, Robot Operating System, OpenCV, Optical flow, Corner finding, Overtaking detection.

Capítulo 1

Introducción

"¡Estudia! No para saber una cosa más, sino para saberla mejor"

Lucio Anneo Séneca. Filósofo, político y escritor romano

SISTEMAS DE ASISTENCIA AL CONDUCTOR

A lo largo del siglo XX los automóviles experimentaron un espectacular aumento en su popularidad y accesibilidad, lo cual llegó de la mano de una nueva causa de muerte: víctimas por accidentes de tráfico. Según estudios de Naciones Unidas, cada año mueren más 1.2 millones de personas en el mundo en dichas circunstancias. [1] Con el objetivo de reducir estas estadísticas, tanto la industria del automóvil como la comunidad científica han investigado las posibles vías de aumento de la seguridad. Actualmente, el campo más importante es el de los sistemas de asistencia al conductor (en inglés *advanced driver assistance systems*, *ADAS*), los cuales no sólo proporcionan información útil sino que incluso pueden realizar acciones sobre el vehículo en caso de peligro. [2]

Los sistemas de asistencia al conductor representan uno de los campos de investigación más activos dedicados a la mejora de la seguridad vial. Aunque su importancia aumenta cada día, estos sistemas llevan siendo desarrollados e introducidos en nuestros vehículos desde hace más de treinta años.

Algunos de los más importantes son los siguientes: sistema antibloqueo de frenos ABS (1978), control de tracción (1986), control de estabilidad (1995), control electrónico de la frenada, control de velocidad de crucero adaptativo (1998), sistema pre-colisión, detector de ángulo muerto (2005), aviso de salida de carril y asistencia de mantenimiento de carril, asistente de visión nocturna, sistema de reconocimiento de señales de tráfico, detector de fatiga en el conductor, asistente al aparcamiento, asistente de arranque en pendiente, sistema de iluminación adaptativa. [3]

VISIÓN POR COMPUTADOR

Muchos de los sistemas anteriormente nombrados han sido desarrollados mediante análisis de imágenes y la Visión por Computador, como el caso nos ocupa en este proyecto.

La visión por computador es una rama de la inteligencia artificial que tiene por objetivo modelar matemáticamente los procesos de percepción visual en los seres vivos y generar programas que permitan simular estas capacidades visuales por un ordenador.

El proceso de visión por computador puede dividirse en seis campos principales: obtención de imágenes, preprocesamiento, segmentación, parametrización, reconocimiento e interpretación, que serán desarrollados en el capítulo 3, 'Estado del arte'.

Este último aspecto, interpretación de la escena, es fundamental y uno de los más críticos a la hora del procesado de imágenes, ya que la correcta interpretación de los datos obtenidos durante los procesos previos proporcionará un resultado óptimo o un conjunto de datos ininteligibles y sin información clara.

En el proyecto que nos ocupa, la extracción de características de la imagen se consigue gracias al método de detección de esquinas Shi-Thomasi y al algoritmo de flujo óptico Lucas-Kanade. [4]

SOFTWARE DEL PROYECTO

El proyecto ha sido realizado bajo el sistema operativo Linux Ubuntu en su versión 12.04 LTS (Long Term-Support) y ROS, un framework de desarrollo de robótica, que se desarrolló originalmente en 2007 por el Laboratorio de Inteligencia Artificial de Standford, para dar soporte a su robot con inteligencia artificial. [5]

Así mismo se ha utilizado OpenCV, una biblioteca libre de visión artificial, y el lenguaje de programación C++.

El software desarrollado ha sido pensado para ser implementado en el vehículo de pruebas de la Universidad Carlos III de Madrid, IVVI 2.0.

Figura 1: Vehículo de pruebas IVVI 2.0 de la UC3M

MEMORIA DEL PROYECTO

En el capítulo uno de la memoria del este proyecto se presenta una introducción de los temas más relevantes que vamos a tratar a lo largo de la misma; en el capítulo dos se mostrarán los objetivos del proyecto.

En el tercer capítulo, se abordarán los recursos más importantes que forman parte del proyecto, mediante su definición, características y aplicación concreta en el proyecto. El capítulo cuatro centra gran parte de la importancia de la memoria ya que en él se desarrollarán y explicarán todas las partes que componen el algoritmo del proyecto.

En el quinto capítulo se tratarán los errores y conclusiones tras la finalización del proyecto. En el sexto capítulo se abordan distintas propuestas de trabajos futuros con la base del presente proyecto. Existe un último apartado bibliográfico con las referencias que aparecen a lo largo de la memoria, tanto de publicaciones y *papers*, como páginas web y libros.

Capítulo 2

Objetivo

"Cuanto más alto coloque el hombre su meta, tanto más crecerá"

Friedrich Schiller. Poeta y dramaturgo

El principal objetivo del proyecto es el de proveer a un vehículo, que disponga del equipo hardware necesario, de un software que permita la detección de adelantamientos a dicho vehículo, con el fin de prestar ayuda al conductor y prevenir posibles situaciones de peligro que puedan darse en la carretera.

Cabe destacar que se trata de un sistema exclusivo de aviso como parte de un proyecto mayor de asistencia a la conducción. En ningún caso el software está diseñado para intervenir en ningún aspecto motriz y/o de funcionamiento del vehículo.

Cuando el software detecte un coche adelantando al vehículo en cuestión, se activará un *flag* que podrá ser utilizado a posteriori como se desee, desde activar un LED de aviso o una pequeña señal acústica hasta dar entrada a un nuevo algoritmo que permita realizar alguna actividad relacionada con el tema.

El algoritmo también proporciona información acerca de la posición aproximada del centro del vehículo, información que será reutilizada en futuras ampliaciones del software.

Capítulo 3

Estado del arte

"Hay dos maneras de diseñar software: una es hacerlo tan simple que sea obvia su falta de deficiencias, y la otra es hacerlo tan complejo que no haya deficiencias obvias"

C.A.R. Hoare. Científico en computación británico

3.1 VISIÓN POR COMPUTADOR Y OPENCV

3.1.1 Visión por computador

La visión por computador tiene como objetivo la extracción de características del mudo que nos rodea a partir de imágenes mediante el uso de un ordenador o computador. Un sistema de visión por computador actúa sobre una representación de la realidad que aporta información sobre su brillo, color, forma, etc. Esta representación de la realidad actúa en forma de imágenes estáticas e imágenes en movimiento (secuencias).

IMAGEN

Una imagen es una función que a cada par de coordenadas (x,y) asocia un valor relacionado con alguna de sus propiedades como puede ser su brillo, por ejemplo. Una imagen acromática, sin color, en la que cada punto tiene asociado exclusivamente información acerca de su brillo se puede representar como una matriz en la que cada punto presenta su nivel de brillo.

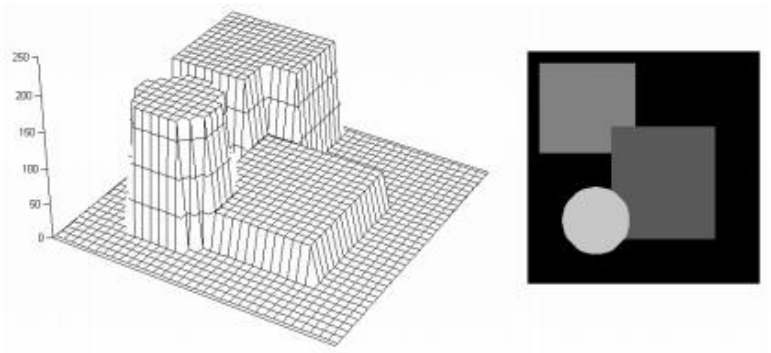

Figura 2: Paso de modelo 3D a 2D según el brillo

En la imagen anterior se puede observar como la representación de las distintas alturas del modelo 3D de la izquierda tiene su representación en 2D asociando a cada altura un brillo.

Una imagen de color RGB se puede representar asociando a cada punto una terna de tres matrices en cada una de las cuales queda representada la intensidad de sus tonos rojo, verde y azul para cada punto de la imagen.

SECUENCIA

Un punto con un brillo suficiente con una frecuencia superior a 25 Hz es percibido como un punto brillante por nuestros sentidos. Cuando se toma una sucesión de imágenes estáticas que se capturan a una frecuencia superior al umbral anteriormente comentado, el sistema visual y de percepción humano no es capaz de distinguir el paso de una imagen a otra y lo interpreta como movimiento. [6]

ETAPAS DE UN SISTEMA DE VISIÓN POR COMPUTADOR

El ser humano capta la luz reflejada en el mundo que nos rodea por medio de los ojos. Esta información circula a través del nervio óptico hasta el cerebro donde es procesada. La visión por computador intenta simular este mismo proceso adaptándolo a sus necesidades. Cada autor define este proceso en varias etapas, pero básicamente pueden clasificarse en las siguientes: [7] [8]

- El primer paso es la **adquisición** de la imagen digital. Para ello se necesitan sensores, cámaras, etc. y la capacidad para digitalizar la señal.
- Una vez la imagen ha sido adquirida, se realiza el **preprocesamiento** de dicha imagen. Se trata de realizar un tratamiento digital mediante filtros y transformaciones geométricas, con el objetivo de eliminar partes indeseables de la imagen o realzar otras zonas de la misma.
- La siguiente etapa es la **segmentación**. Su objetivo es dividir la imagen en las partes que la constituyen o los objetos que la forman. La salida de esta etapa nos proporcionará los puntos frontera que delimitan la zona de interés o todos los puntos que definen dicha forma. La representación por frontera es apropiada cuando el objetivo se centra en las características de la forma externa como esquinas o bordes. La representación por regiones es apropiada cuando se desea adquirir información interna como la textura.
- La **parametrización** se dedica a extraer rasgos que producen alguna información cuantitativa de interés o rasgos básicos para diferenciar objetos de un tipo de otro.
- El siguiente paso es el **reconocimiento**, proceso que asigna una etiqueta basada en las preferencias del usuario y sus necesidades.

- La última etapa es la interpretación, que lleva a asignar un significado al conjunto de objetos, puntos y/o características obtenidas.

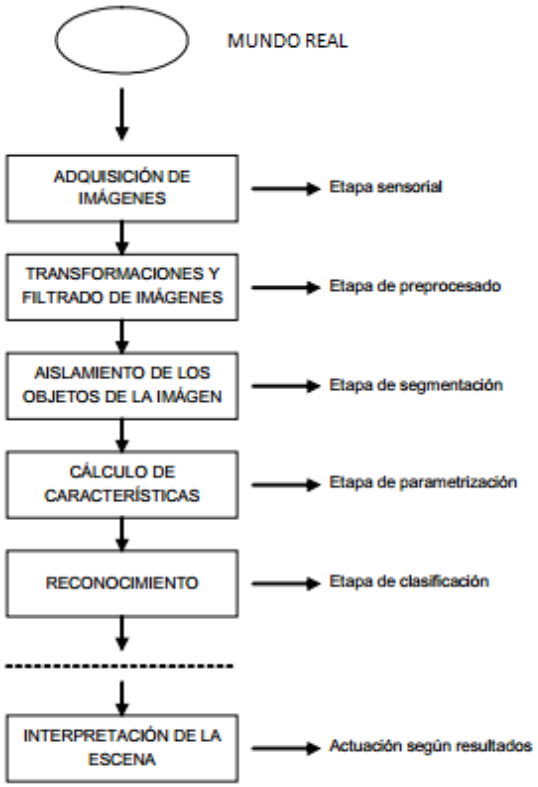

Figura 3: Etapas de un sistema de visión por computador

COMPONENTES DE UN SISTEMA DE VISIÓN POR COMPUTADOR

Los componentes hardware mínimos requeridos para completar el sistema son los siguientes: [6] [7]

- Un **sensor óptico** para capturar la imagen: puede ser una cámara de vídeo, una cámara fotográfica, una cámara digital, un escáner, etc. La elección de este dispositivo suele estar determinada por la aplicación diseñada.
- Una **tarjeta de adquisición de imagen** que permite digitalizar la señal de vídeo entregada por el sistema anterior.
- Un **ordenador** que almacene las imágenes y que ejecute los algoritmos de preprocesado, segmentación, parametrización y clasificación.
- Como componente adicional, se puede instalar un **monitor de vídeo** que permita visualizar tanto las imágenes captadas como los resultados del procesamiento de dichas imágenes.

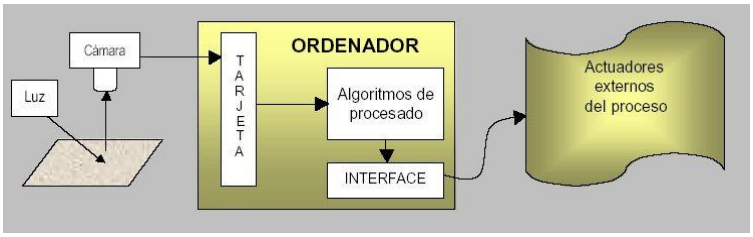

Figura 4: Componentes de un sistema de visión por computador

3.1.2 Biblioteca de visión OpenCV

OpenCV es una biblioteca libre de visión artificial. Su desarrollo ha ido de la mano de aplicaciones desde sistemas de detección de movimiento hasta control de procesos mediante reconocimiento de objetos. Se permite que sea usada libremente para propósitos, tanto comerciales como de investigación, gracias a su licencia *BSD (Berkley Software Distribution)* .[9]

La biblioteca ha sido escrita en lenguaje *C* y *C++* y funciona bajo *Linux*, *Android*, *Windows*, *iOS* y *MacOS X*, aunque en la actualidad se siguen desarrollando trabajos para su adaptación a *Python*, *Ruby*, *Matlab* y otros lenguajes.

OpenCV fue diseñado para lograr una alta eficacia computacional centrándose en aplicaciones en tiempo real. Uno de sus objetivos es proporcionar una infraestructura de visión por computador sencilla de usar y que permita construir aplicaciones sofisticadas rápidamente. Dicha biblioteca contiene más de 500 funciones implementadas que abarcan diversas áreas de visión, incluyendo mapeado médico, seguridad, interfaz de usuario, calibración de cámaras, visión estéreo y robótica.

Gracias a la licencia libre de la que dispone, se ha creado una gran comunidad de usuarios y desarrolladores que incluye trabajadores de las mayores compañías (IBM, Microsoft, Intel, Sony, Siemens, Google, Toyota, etc.) y de centros de investigaciones tales como Stanford, MIT y Cambridge. De la misma forma, se han creado plataformas de ayuda para *OpenCV* como *Yahoo* [10] y *Stack Overflow* [11], por nombrar algunas de las más importantes, con miles de desarrolladores tanto profesionales como no profesionales. [4]

HISTORIA

Esta librería nació en 1999 en Intel gracias Gary Bradski. La primera versión alfa de *OpenCV* apareció en público en el *IEEE Conference on Computer Vision and Pattern Recognition* [12] en el año 2000. Posteriormente, del 2001 al 2005 salieron 5 versiones betas, y la primera versión oficial no beta (la versión 1.0) surgió en 2006. Un par de años después, en 2008, *OpenCV* recibió el apoyo de la compañía Willow Garage, que actualmente sigue encargándose de su desarrollo. [4]

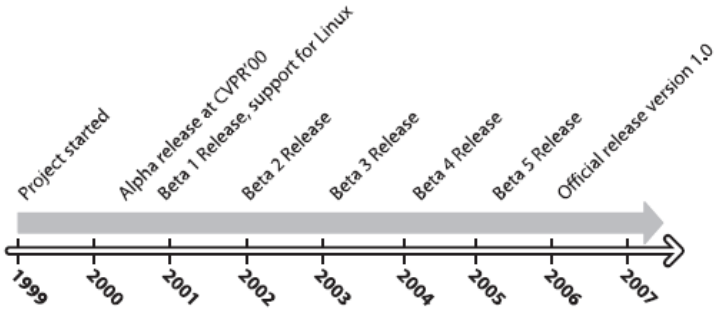

Figura 5: Timeline histórico de OpenCV

ESTRUCTURA

OpenCV presenta una estructura modular, lo que significa que el paquete incluye un gran número de librerías estáticas y compartidas. Los módulos disponibles son los siguientes: [13] [14]

- **core**: un módulo compacto que define las estructuras básicas de datos, incluyendo los arrays multidimensionales tipo *Mat* (del cual serán las imágenes con las que se trabaja en este proyecto) y funciones básicas usadas por el resto de los módulos.

- **imgproc**: es un módulo de procesado de imágenes que incluye filtro lineal y no lineal, transformaciones geométricas y conversiones de color.

- **video**: módulo de análisis de vídeos que incluye estimación de movimiento, diferenciación y extracción de fondos y algoritmos de seguimiento de objetos.

- **highgui**: una interfaz de captura de vídeo fácil de usar.

- **calib3d**: calibración de cámaras y reconstrucción en 3D desde vistas de múltiples ángulos.
- **objdetect**: detección de objetos usando clasificadores en cascada.
- **ML**: modelos estadísticos y algoritmos de clasificación para su uso en aplicaciones de visión por computador.
- Existen otros módulos como features2d, GPU, flan, stitching, nonfree.

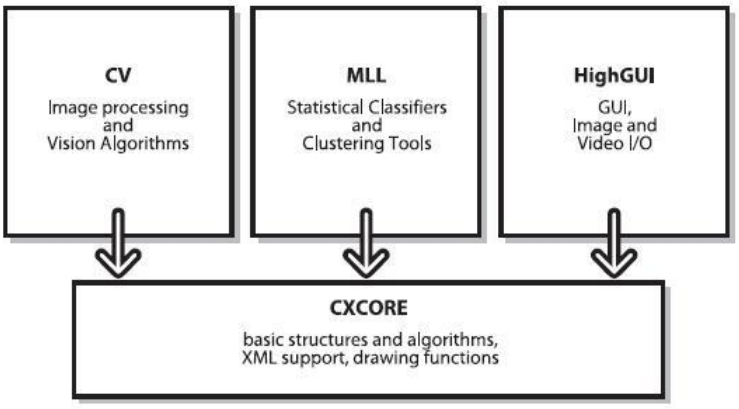

Figura 6: Estructura modular de OpenCV

3.2 SISTEMA OPERATIVO LINUX UBUNTU 12.04 LTS

Ubuntu es un sistema operativo basado en GNU/Linux, distribuido como software libre [15], principalmente enfocado a su facilidad de uso e instalación, libertad de los usuarios y lanzamientos regulares cada seis meses.

Su nombre proviene del término Ubuntu, una regla ética sudafricana enfocada a la lealtad de las personas y las relaciones entre éstas, encabezado por el obispo Desmond Tutu, Premio Nobel de la Paz 1984. Está orientado al usuario novel con un fuerte enfoque en la facilidad de uso y en mejorar la experiencia de usuario.

La filosofía de Ubuntu se basa en los siguientes principios:

- Debe ser capaz de utilizar todo el software independientemente de su formación.

- El usuario debe tener la libertad de descargar, ejecutar, copiar, distribuir, estudiar, compartir, cambiar y mejorar su software sin tener que pagar derechos de licencia.

- Debe ser capaz de utilizar su software en el idioma de su elección (más de 130). [16]

Su patrocinador principal es la compañía británica Canonical, la cual ofrece el sistema de manera gratuita y se financia por medio de servicios vinculados al sistema operativo y mediante la venta de soporte técnico. Además de la aportación de la empresa, al mantenerlo como un SO libre y gratuito, la gran cantidad de desarrolladores de la comunidad permite su continua mejora.

VERSIONES Y LANZAMIENTOS

Ubuntu ofrece, a grandes rasgos: [17]

- Cuatro versiones con software personalizado: Ubuntu, Kubuntu, Xubuntu, Edubuntu.

- Cuatro versiones personalizadas para distintos tipos de usuarios: Desktop CD, Alternate CD, Server CD o DVD.

- Tres versiones para tres tipos de ordenadores.

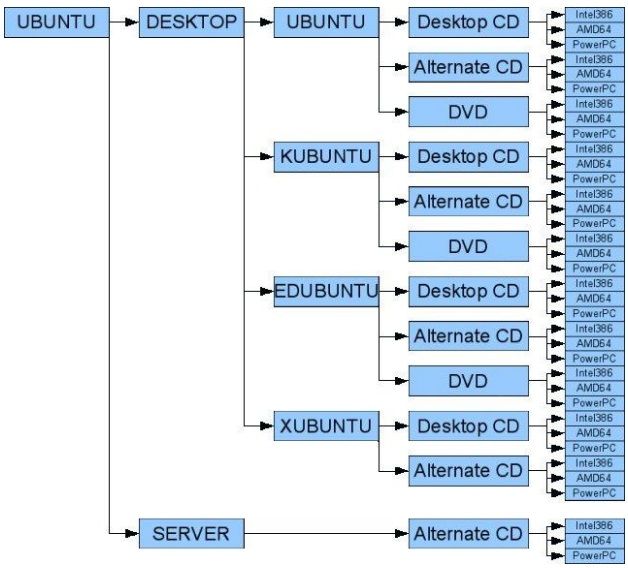

Figura 7: Esquema de las versiones de Ubuntu

Para realizar este proyecto se ha optado por la versión Ubuntu Desktop CD para Intel x86.

En cuanto a los lanzamientos, los no LTS se liberan cada seis meses y Canonical proporciona soporte técnico y actualizaciones de seguridad durante nueve meses, a partir de la versión 13.04 en adelante. Las versiones LTS aparecen en abril cada dos años y ofrecen un soporte técnico de cinco años para la versión de escritorio (Desktop) y servidor (Server) desde la fecha de lanzamiento.

El código de cada lanzamiento se especifica con dos números de dos cifras: la primera indica el año y la segunda el mes de lanzamiento. Desde sus inicios, cada uno de los ciclos de desarrollo de Ubuntu ha tenido un nombre en clave compuesto por un animal y un adjetivo relativo al mismo que empiece por su misma letra (Trusty Tahr). Además, desde la versión 6.06 (Dapper Drake), estos nombres han seguido una progresión por orden alfabético.

Para realizar este proyecto se ha optado por la versión 12.04 LTS, la última versión con largo soporte técnico en el momento en que éste comenzó.

En la actualidad, la última versión LTS es la 14.04.

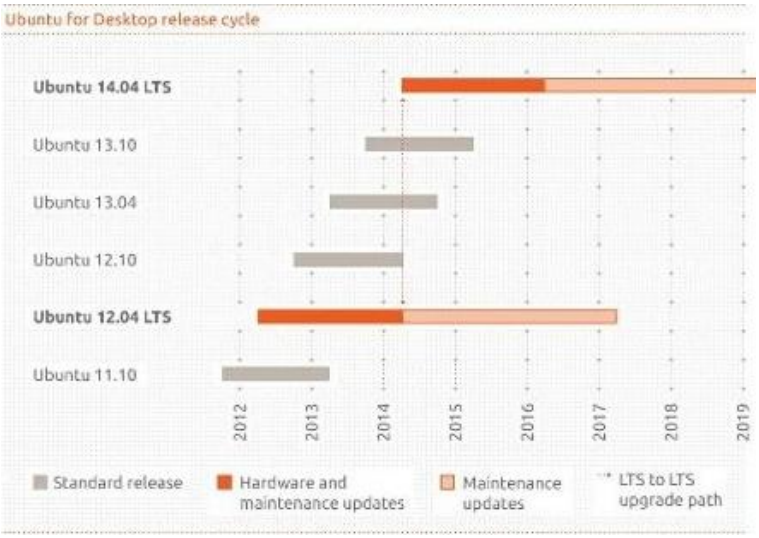

Figura 8: Últimos lanzamientos Ubuntu y su soporte

Según al dispositivo que se disponga, existen varias versiones para muchos de ellos: Ubuntu Desktop, ordenador de escritorio; Ubuntu Phone, teléfonos inteligentes; Ubuntu Tablet, tabletas portátiles; Ubuntu TV, televisores inteligentes; Ubuntu for Android, teléfonos inteligentes Android; Ubuntu Server, servidores; y Ubuntu Business Desktop, orientada al sector empresarial.

3.3 ROS Y EL LENGUAJE DE PROGRAMACIÓN C++

3.3.1 ¿Qué es ROS?

ROS es una plataforma de desarrollo de código abierto para sistemas robóticos, que proporciona los servicios propios de un sistema operativo (a pesar de que no lo es, ya que en nuestro proyecto lo haremos funcionar bajo Linux), incluyendo: abstracción de hardware, control de dispositivos a bajo nivel, implementación de funcionalidades de uso habitual, intercambio de información entre procesos propios y manipulación de paquetes o proyectos. También ofrece herramientas y librerías para obtener, construir, escribir y hacer funcionar software entre varios ordenadores.

Desde su creación, *ROS* se ha diferenciado (entre otros aspectos) por facilitar el intercambio de software entre aficionados y profesionales del mundo de la robótica debido a su enfoque didáctico y abierto, lo que ha influido en que en la actualidad exista una gran red de desarrolladores y colaboradores a lo largo del mundo.

A diferencia de la mayoría de las plataformas de desarrollo robótico existentes, *ROS* pretende dar una solución integral al problema de desarrollo de robots aportando soluciones en las siguientes áreas:

- Reutilización e integración de robots y dispositivos encapsulando estos tras interfaces estables y manteniendo las diferencias en archivos de configuración.
- Algoritmos de robótica con bajo acoplamiento: desde algoritmos de bajo nivel de control, cinemática, hasta algoritmos de alto nivel como planificación o aprendizaje.
- Simulación de sistemas robóticos en mundos virtuales con dinámicas de sólidos rígidos.
- Herramientas de desarrollo, despliegue y monitorización de sistemas robóticos.

- Mecanismo de comunicaciones (middleware) distribuido entre nodos del sistema robótico (a continuación se explicará en detalle dicho sistema de comunicación).

ROS puede ser ejecutado sobre ordenadores trabajando bajo Unix aunque también puede encontrarse soporte para otras plataformas como *Fedora* y *Gentoo* gracias a su extensa red de colaboradores.

Su objetivo principal es dar apoyo a la reutilización de código en investigaciones y desarrollo robótico y es un marco de trabajo entre procesos (en adelante conocidos como nodos) que generan ejecutables. Estos nodos pueden estar agrupados en paquetes o pilas, que pueden ser fácilmente compartidos y distribuidos.

En apoyo a este objetivo principal de colaboración, existen otros aspectos que hacen de *ROS* un *framework* especial:

Un nodo es cualquier pieza de software del sistema (desde un algoritmo SLAM hasta un driver para el manejo de un motor). El objetivo de este sistema es doble:

1. Encapsulado/abstracción/reutilización de software.
2. Ubicuidad, es decir, independencia de donde este nodo está localizado (un sistema robótico puede tener muchos procesadores).

Estos nodos se comunican entre ellos mediante mecanismos de paso de mensajes RPC o Publish/Subscribe, Servicelookup, etc. Permite crear arquitecturas *P2P* de componentes robóticos distribuidos. [18] [19]

Tanto el núcleo del sistema *ROS* como los útiles, herramientas y bibliotecas son creados y actualizados regularmente con una distribución de *ROS*. Estas distribuciones son similares a cualquiera de *Linux*. Las distribuciones históricas de *ROS* son las siguientes: [20] [21] [22]

- ROS Box Turtle (Marzo, 2010)
- ROS C Turtle (Agosto, 2010)
- ROS Diamondback (Marzo, 2011)
- ROS Electric (Agosto, 2011)
- ROS Fuerte (Abril, 2012)
- ROS Groovy (Diciembre, 2012)
- ROS Hydro Medusa (Septiembre, 2013)
- ROS Indigo Igloo (Mayo 2014)

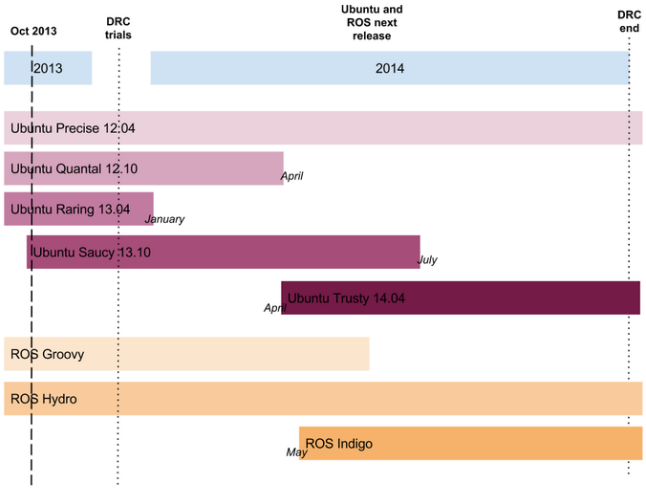

Figura 9: Timeline histórico de distribuciones de ROS y Ubuntu

Pocas cosas en *ROS* son auténticamente nuevas a nivel técnico; en realidad parece estar inspirado en muchos *frameworks* de desarrollo robótico exitosos como *Placer/Stage/Gazebo, Yarp, Orocos, Carmen, Orca, OpenRave, open-RTM* (todos ellos *OpenSource*) e incluso en muchos aspectos se asemeja a tecnologías más lejanas como *JADE* (*Sistemas Multi-Agentes*). Sin embargo, *ROS* ha logrado agrupar muchas de las mejores características de todos estos proyectos dando una solución integral y muy uniforme al problema de desarrollo de sistemas robóticos. Se caracteriza además por ser una plataforma multi-lenguaje (*c++, phyton, java*).

3.3.2 Desarrollar un proyecto básico en ROS

CREAR UN PAQUETE

El software de *ROS* se encuentra organizado en paquetes (en inglés, *packages*). Un paquete puede contener nodos, librerías independientes, archivos de configuración, o cualquier otro aspecto que constituya un módulo útil para ser reutilizado. El principal objetivo de estos paquetes es conseguir la mezcla óptima entre funcionalidad y facilidad de uso, de tal manera que su uso no sea complejo desde otro software. [23]

Todos los paquetes de *ROS* constan de los mismos archivos: *manifests*, *CMakeLists.txt*, *mainpage.dox* and *Makefiles*.

Utilizando el comando `roscreate`, se ahorra la tediosa tarea de crear un nuevo paquete a mano y se eliminan los errores comunes derivados de construir dichos archivos.

Antes de introducir dicho comando, se debe acceder a la carpeta raíz donde se creará el paquete, que debe estar en `/fuerte_workspace/[carpeta_raíz]`. A la vez que se crea el paquete, se le asignará un nombre y sus dependencias (éstas pueden ser modificadas posteriormente si fuera necesario): [24]

```
cd ~/fuerte_workspace/[carpeta_raíz]
roscreate-pkg [package_name] [dep1] [dep2] [dep3] …
```

Tras ello aparecerá una imagen como la siguiente:

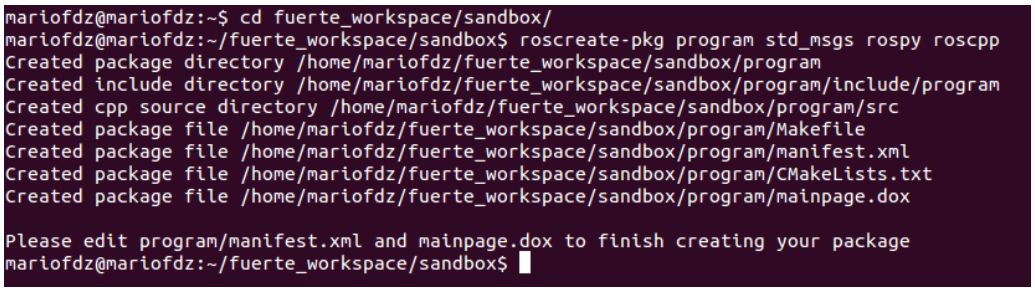

Figura 10: Creación de un paquete ROS

CONSTRUIR UN PAQUETE

Una vez todas las dependencias del sistema han sido instaladas, se construye el paquete que acaba de ser creado.

Para ello se utiliza el comando `rosmake` que, además de construir el paquete seleccionado, construye todos los paquetes que dependen de él (`roscpp`, `std_msgs`, etc).[25]

Se introduce el siguiente comando en el terminal:

```
rosmake [package]
```

Si no ha habido errores, veremos el siguiente mensaje.

```
mariofdz@mariofdz:~$ rosmake program
[ rosmake ] rosmake starting...
[ rosmake ] Packages requested are: ['program']
[ rosmake ] Logging to directory /home/mariofdz/.ros/rosmake/rosmake_output-2014
0915-173907
[ rosmake ] Expanded args ['program'] to:
['program']
[rosmake-0] Starting >>> std_msgs [ make ]
[rosmake-1] Starting >>> roslang [ make ]
[rosmake-1] Finished <<< roslang  No Makefile in package roslang
[rosmake-1] Starting >>> rospy [ make ]
[rosmake-0] Finished <<< std_msgs  No Makefile in package std_msgs
[rosmake-0] Starting >>> roscpp [ make ]
[rosmake-0] Finished <<< roscpp  No Makefile in package roscpp
[rosmake-1] Finished <<< rospy  No Makefile in package rospy
[rosmake-1] Starting >>> program [ make ]
[rosmake-1] Finished <<< program [PASS] [ 5.81 seconds ]
[ rosmake ] Results:
[ rosmake ] Built 5 packages with 0 failures.
[ rosmake ] Summary output to directory
[ rosmake ] /home/mariofdz/.ros/rosmake/rosmake_output-20140915-173907
mariofdz@mariofdz:~$
```

Figura 11: Construcción de un paquete ROS

CREAR UN NODO

Un nodo no es más que un archivo ejecutable dentro de un paquete de *ROS*. Los nodos de *ROS* utilizan una biblioteca cliente para comunicarse con otros nodos, así como publicar y/o suscribirse a un *topic*.

Figura 12: Nodos de ROS

La librería cliente de *ROS* permite al usuario escribir los nodos en diferentes lenguajes; en nuestro caso se usó `roscpp` (lenguaje *C++*).[26] [27] [28]

Dentro del proyecto, en la carpeta *src*, se crea un archivo (*main.cpp*) que contiene el código del algoritmo, tanto la función principal `main` como la función `imageCallback`.

El comando `rosnode` permite listar y adquirir información de los nodos. En el Anexo II se encuentra el resumen de *ROS* donde aparece cómo acceder a dicha información.

CONSTRUIR UN NODO

El último paso es crear el ejecutable propio del nodo que acaba de ser creado. Para ello en primer lugar se modificará el archivo *CMakeLists.txt*, añadiendo la línea de código que permite crear el ejecutable del nodo. En nuestro caso:

```
rosbuild_add_executable (overtaking src/main.cpp)
```

Por último se volverá a construir el paquete de tal forma que se incluyan los cambios generados en el archivo *main.cpp* y *CMakeLists.txt* y se cree el ejecutable, con la orden `rosmake`.

3.3.3 Aspectos de ROS aplicados al proyecto

ROS TOPICS

Antes conviene explicar qué es un mensaje: un mensaje es el medio que tiene *ROS* para comunicar nodos entre sí.

Los *topics* o temas son los nombres que identifican el contenido de un mensaje. Pueden transmitirse de dos formas: publicados (*publisher*) o suscritos (*subscriber*).

En el proyecto que nos incumbe, el *topic* será *usb_cam/image_raw* y nuestros nodos serán *usb_cam* (el cual publicará) y *overtaking* (el cual se suscribirá). En la siguiente imagen puede observarse esta relación:

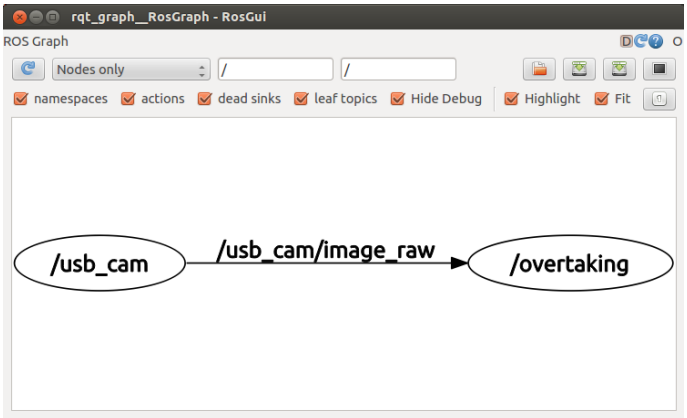

Figura 13: Relación entre nodos y topic del proyecto

Esta es una de las verdaderas ventajas que ofrece *ROS*: al realizar un nuevo proyecto, no es necesario decidir o saber a priori a qué dispositivo (cámara, USB, etc) ha de conectarse el programa, basta con saber qué tipo de dato se espera recibir (imágenes, por ejemplo) y suscribirse al nodo creado a tal fin. Esto ahorra tiempo y permite reciclar el trabajo ya realizado (por uno mismo o por otro usuario).

Puede haber al mismo tiempo varios publicadores o suscriptores que coincidan en un mismo tema y el mismo nodo puede publicar y/o suscribirse a varios temas. En general, los nodos no son conscientes de con qué otro nodo están comunicados, simplemente lo son del tema que les llega o que publican. [29] [30]

El comando `rostopic` permite gran variedad de acciones al usuario a la hora de adquirir información de los *topics* (mostrar los datos de un tema, listar aquellos que se encuentran en funcionamiento, publicar directamente datos en ellos). Se puede también mostrar las conexiones entre nodos y temas utilizando el paquete *rqt_graph*, que proporciona una imagen como la de la figura anterior. Todas estas funcionalidades pueden consultarse en el Anexo II. [31]

NODO SUSCRIPTOR (SUBSCRIBER)

En el proyecto, se creó el código sobre la base de un nodo de suscripción, ya que nuestro algoritmo debía recibir las imágenes que el nodo *usb_cam* publicaba. La estructura de este tipo de nodos aparece en los tutoriales de *ROS* en la web. [32] La explicación del código se verá en el capítulo 4, 'Desarrollos', en su apartado correspondiente, pero en este momento conviene definir qué ocurre al lanzar un nodo así:

- Primero se inicializa el sistema de *ROS*.
- A continuación, se suscribe al tema requerido (en nuestro caso *usb_cam/image_raw*, como se vio antes).
- Entra en un bucle a la espera de la llegada del mensaje.
- Cuando el mensaje llega, se llama a la función *imageCallback* (denominada así en nuestro programa).

USO Y ACCESO A DATOS DE ROS. BAGS

Bags o bolsas son un formato para guardar y reproducir datos de un mensaje de *ROS*. Presentan una gran cantidad de herramientas que han sido desarrolladas y que permiten almacenar, analizar, procesar y visualizarlas.[33]

En este proyecto se han utilizado *bags* y sus funcionalidades para realizar las pruebas necesarias del algoritmo, como se explicará en el capítulo 5.

La forma de acceder a dichos archivos se consigue a través del comando `rosbag` que permite: guardar *topics* publicados, obtener información de un archivo *bag* o reproducirlo. En el Anexo II se pueden consultar estas funcionalidades.[34]

3.3.4 Lanzar el proyecto

NODO *ROSCORE* DE ACTIVACIÓN DE ROS

roscore es una colección de nodos y programas que son requeridos para cualquier sistema basado en *ROS*. Siempre se debe tener *roscore* operando para permitir la comunicación entre otros nodos.

Para lanzarlo basta con introducir el siguiente comando:

```
roscore
```

Una vez lanzado, en terminal aparecerá lo siguiente:

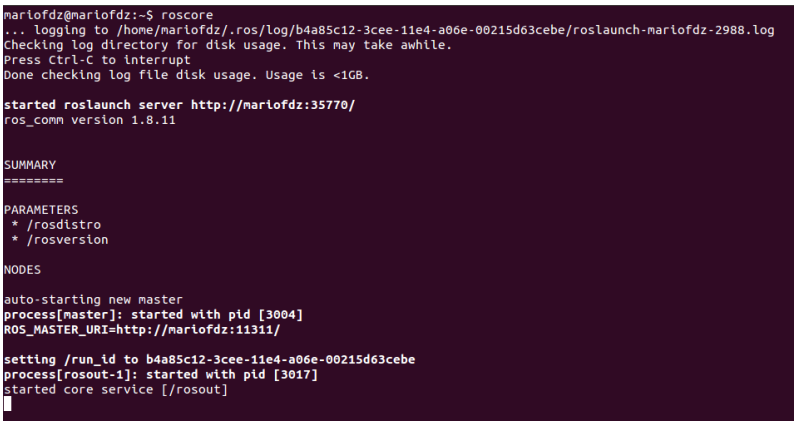

Figura 14: Activación del nodo roscore

También se activará el nodo *rosout* que permite detener el sistema. [35]

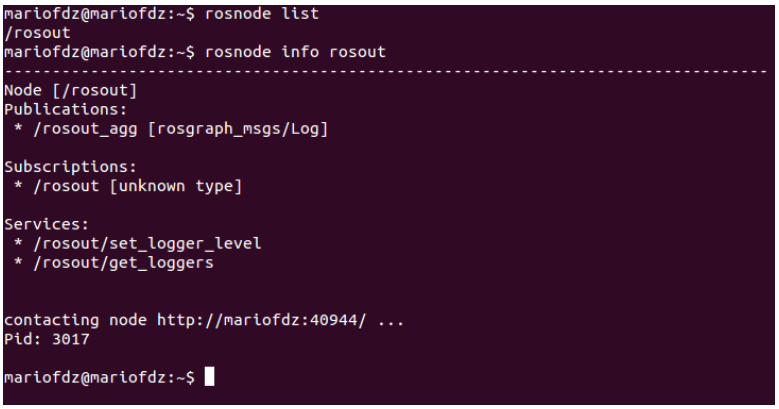

Figura 15: Información del nodo rosout

NODO *usb_cam* DE ACCESO A LA CÁMARA

Una vez iniciado el *roscore*, es necesario lanzar el nodo *usb_cam_node* del paquete *usb_cam*. Este nodo conecta con las cámaras USB estándar usando *libusb_cam* y publica imágenes como *sensor_msgs::Image*. Para ello usa librerías que permiten el transporte de imágenes comprimidas.[36]

Antes de lanzar dicho nodo, es necesario ajustar el parámetro *pixel_format* a '*yuyv*'. Se realiza en el terminal mediante el siguiente comando:

```
rosparam set/usb_campixel_format yuyv
```

A continuación se lanza el nodo:

```
rosrun usb_cam usb_cam_node
```

donde *usb_cam* es el nombre del paquete y *usb_cam_node* es el nodo de dicho paquete que permite el transporte de las imágenes de la cámara asociada. Al lanzarlo, también se publica la velocidad en frames/segundo, tal y como se puede ver a continuación:

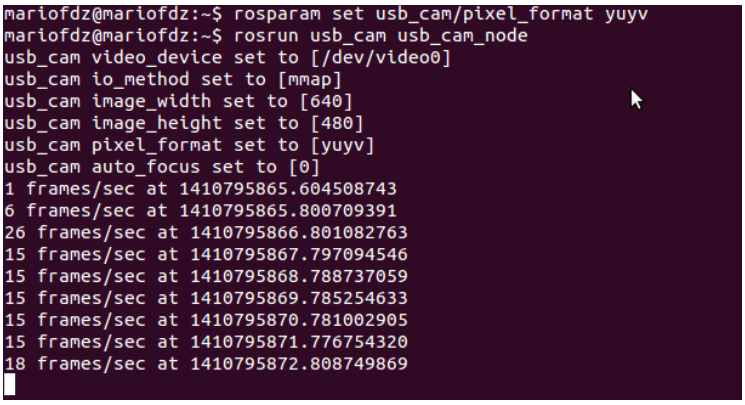

Figura 16: Activación del nodo usb_cam de acceso a la cámara

NODO *cam_subscriber* DE LANZAMIENTO DEL SOFTWARE

Una vez disponemos del nodo *usb_cam_node* de transporte de imágenes activado, solo es necesario lanzar nuestro propio algoritmo:

```
rosrun cam_subscriber overtaking
```

Cuando se encuentra en funcionamiento el software desarrollado, los nodos activados son los esperados:

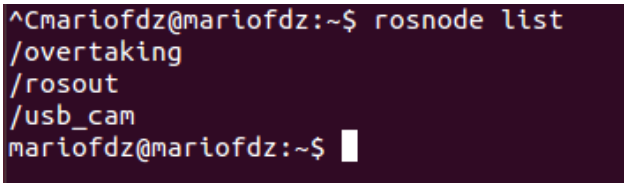

Figura 17: Nodos activos con el software en funcionamiento

3.3.5 ROS y OpenCV

Hasta ahora se ha visto y comentado ROS y OpenCV por separado. Para trabajar con ellos de forma conjunta es necesario realizar ciertos ajustes:

AGREGAR OPENCV A UN PAQUETE

Una vez instalados *OpenCV* y *ROS* y creado el paquete del proyecto como se explicó anteriormente, es necesario ajustar ciertos archivos para que *ROS* sepa que el

paquete necesita *OpenCV* para correr y compilar. Esto se consigue haciendo lo siguiente: [37]

- En primer lugar, hay que modificar el archivo CMakeLists.txt añadiendo las siguientes líneas:

```
find_package(OpenCV REQUIRED)
target_link_libraries(your_node ${OpenCV_LIBS})
```

- En segundo lugar, y siempre que no se hiciera al crear el paquete, hay que incluir la dependencia adecuada en el archivo manifest.xml:

```
<rosdep name="opencv2"/>
```

USAR OPENCV EN UN PROYECTO DE ROS

Es necesario convertir las imágenes de *ROS* a imágenes de *OpenCV* (formato *cv::Mat*), con las que se trabaja en el proyecto, y viceversa, usando *cv_bridge*. [38]

ROS pasa las imágenes en su propio formato de mensaje llamado *sensor_msgs/Image*. CvBridge es una librería de *ROS* que proporciona una interfaz entre *ROS* y *OpenCV*, y que se encuentra en el paquete *cv_bridge* de la pila o *stack visión_opencv*, el cual también contiene el paquete image_geometry, una colección de métodos para interactuar sobre la geometría de imágenes y píxeles. [39]

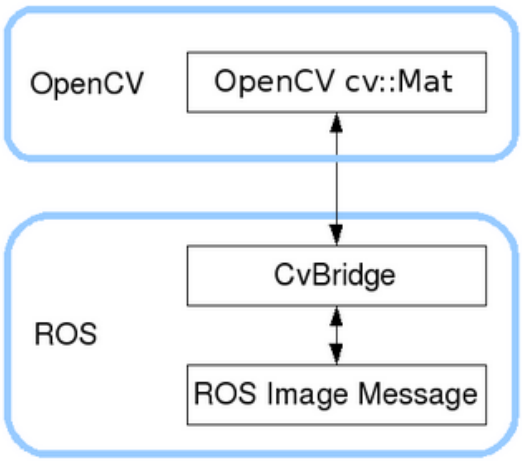

Figura 18: Esquema de las versiones de Ubuntu

CONVERTIR MENSAJES IMAGEN DE ROS A IMÁGENES OPENCV

CvBridge define una clase CvImage que contiene entre sus atributos una imagen OpenCV de tipo cv::Mat. Dicha clase está definida de la siguiente forma:

```
namespace cv_bridge {

class CvImage
{
public:
  std_msgs::Header header;
  std::string encoding;
  cv::Mat image;
};

typedef boost::shared_ptr<CvImage> CvImagePtr;
typedef boost::shared_ptr<CvImage const> CvImageConstPtr;

}
```

Al convertir un mensaje tipo sensor_msgs/Image en tipo CvImage, CvBridge reconoce dos casos distintos de uso:

- Si se modifican los datos. Es necesario hacer una copia de los datos de un mensaje de *ROS*.

- Si no se modifican los datos: Se comparten los datos de dicho mensaje de *ROS* en lugar de copiarlos.

CvBridge proporciona las siguientes funciones para convertir los mensajes a *CvImage*:

```
// Case 1: Always copy, returning a mutable CvImage
CvImagePtr toCvCopy(const sensor_msgs::ImageConstPtr& source,
                    const std::string& encoding =
std::string());
CvImagePtr toCvCopy(const sensor_msgs::Image& source,
                    const std::string& encoding =
std::string());

// Case 2: Share if possible, returning a const CvImage
CvImageConstPtr toCvShare(const sensor_msgs::ImageConstPtr&
source,
```

```
                                const std::string& encoding =
std::string());
CvImageConstPtr toCvShare(const sensor_msgs::Image& source,
                                const boost::shared_ptr<void
const>& tracked_object,
                                const std::string& encoding =
std::string());
```

El primer argumento de entrada es un puntero a la imagen. El segundo es la codificación deseada para dicha imagen. Esta codificación es de tipo *string*, y se puede elegir entre las siguientes: [40]

- *mono8*: CV_8UC1, imagen en escala de grises de 8 bits.
- *mono16*: CV_16UC1, imagen en escala de grises de 16 bits.
- *bgr8*: CV_8UC3, imagen con 3 canales en orden azul, verde y rojo de 8 bits.
- *rgb8*: CV_8UC3, imagen con 3 canales en orden rojo, verde y azul de 8 bits.
- *bgra8*: CV_8UC4: imagen BGR con canal alfa de 8 bits.
- *rgba8*: CV_8UC4: imagen RGB con canal alfa de 8 bits.

CvBridge también reconoce las siguientes codificaciones Bayer: *bayer_rggb8*, *bayer_bggr8*, *bayer_gbrg8* y *bayer_grbg8*. [40]

3.3.6 Lenguaje de programación C++

LENGUAJE DE PROGRAMACIÓN

Un lenguaje de programación es un sistema estructurado y diseñado que permite al usuario comunicarse con un ordenador con el fin de que realice una tarea específica. Contiene un conjunto de acciones consecutivas que el ordenador debe ejecutar. Cada lenguaje posee su propia sintaxis y medios de prueba, depuración y compilación.

Se pueden clasificar en:

- Lenguajes de programación de bajo nivel: son aquellos utilizados para comunicarse y controlar directamente el hardware de un sistema o una máquina, ordenando las operaciones fundamentales que se deben realizar.

- Lenguaje de programación de alto nivel: se caracterizan por que permiten expresar las funciones programadas de forma adecuada a la comprensión humana, en lugar de la sintaxis más cercana a la máquina.

LENGUAJE DE PROGRAMACIÓN C++

C++ es un lenguaje imperativo derivado del C. Puede considerarse como un subconjunto aunque es posible escribir C que resulte erróneo o ilegal en C++. En cualquier caso, desde su creación se optó por mantener una alta compatibilidad con su antecesor por dos motivos principales: en primer lugar para facilitar la transición a los desarrolladores de C clásico y para poder reutilizar la gran cantidad de código.

Se trata de un lenguaje clasificado dentro de la programación orientada a objetos, *OOP* (en inglés *object-oriented programming*) que permite definir clases que describe los atributos y métodos de los objetos similares entre sí (definidos dentro de la misma clase).

Como se comentó anteriormente, *ROS* ha sido optimizado para C++ y este es el motivo por el que se decidió realizar el proyecto en dicho lenguaje.

Capítulo 4

Desarrollos

"El buen código es su mejor documentación"

Steve McConnell. Escritor de ingeniería de software

4.1 DESCRIPCIÓN DEL ALGORITMO FINAL

A continuación se explicará con detalle cada parte del código: qué funciones aparecen, qué conceptos *OpenCV* y *ROS* intervienen en cada apartado y la explicación de cada uno de ellos.

El código se divide en dos funciones: una función principal `main`, en la que, entre otros aspectos, se define la suscripción a las imágenes publicadas a través de mensajes *ROS*, y una función llamada `imageCallback` que será llamada desde el `main`, donde se realizan todas las operaciones de procesado de la imagen.

4.1.1 Adquisición de imágenes

El algoritmo adquiere las imágenes que posteriormente procesa por medio del *topic usb_cam/image_raw*. Como se explicó en el apartado 3.3 del tercer capítulo, el nodo creado para el software del proyecto (*cam_subscriber*) se suscribe al tema anterior que publica el nodo *usb_cam*.

Esto se consigue por medio del siguiente código:

```
image_transport::ImageTransport it(nh);

image_transport::Subscriber sub = it.subscribe
("usb_cam/image_raw", 1, imageCallback);
```

donde *usb_cam/image_raw* es el *topic* al que se suscribe nuestro nodo e `imageCallback` es la función en la que se procesan los mensajes recibidos.

4.1.2 Codificación en escala de grises

Tras subscribirse el software al *usb_cam* y obtener las imágenes que éste publica (en nuestro caso las obtenidas por la cámara embarcada en el turismo), se asigna un puntero de imagen (*cv_ptr*) por medio de la función *toCvCopy* [40] que permite codificar directamente la copia de la imagen a escala de grises. Esto es necesario para el siguiente paso, la detección de esquinas usando el método Shi-Tomasi.

La función descrita es la siguiente:

```
CvImagePtrtoCvCopy(const
sensor_msgs::ImageConstPtr& source,const
std::string& encoding = std::string());
```

donde el primer parámetro es el mensaje que envía ROS y recibe el programa y el segundo parámetro es la codificación necesaria. Las posibles codificaciones son: mono8 y mono16 de 1 canal; bgr8 y rgb8 de 3 canales; bgra8 y rgba8 de 4 canales. La función queda de la siguiente forma:

```
cv_ptr = cv_bridge::toCvCopy(msg, "mono8");
```

4.1.3 Creación de una región de interés

Para mejorar el tiempo de procesado de cada *frame* se optó por crear una región de interés sobre la que trabajar. Esta zona de interés o ROI (*region of interest*) se ajustó como un rectángulo desde la esquina superior izquierda, origen de coordenadas para OpenCV, hasta un 50% del tamaño máximo de alto de la imagen original y hasta un 30% del ancho de dicha imagen. Esto es:

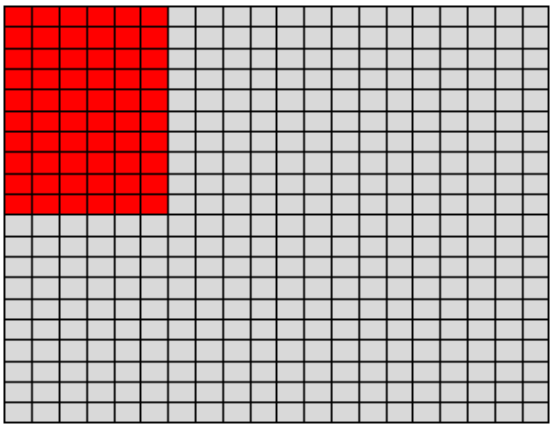

Figura 19: Asignación de ROI respecto a la imagen original

Para este ajuste se utilizan dos parámetros, *param_col* y *param_row*, que deberán ser calibrados para cada cámara y la posición en la que se embarque en el vehículo. En el proyecto toman valores 0.3 y 0.5 respectivamente de acuerdo a las

pruebas con el archivo *bag* proporcionado, como se explicará más adelante en el capítulo 6 de la memoria.

Para realizarlo se comienza por calcular el tamaño de la imagen que llega al algoritmo procedente del *topic* de *ROS*; de esta forma siempre se escogerá un ROI de la misma proporción, independientemente del tamaño original de las imágenes a las que se suscriba el programa. El código es el siguiente:

```
int rows = image.rows;
int cols = image.cols;
cv::Size s = image.size();
rows = s.height;
cols = s.width;
```

Una vez se obtiene la altura definida en la variable `rows` (filas) y la anchura definida en `cols` (columnas), se procede a definir el rectángulo que dará cabida al ROI:

```
int param_col = 0.3;
int param_row = 0.5;
cv::Rect region_of_interest = Rect (0, 0,
param_col*cols, param_row*rows);
cv::Mat image_roi = image(region_of_interest);
```

donde los dos primeros parámetros de la función `Rect` [41] son las coordenadas x e y desde las que se define la altura y la anchura, parámetros tercero y cuarto de dicha función.

4.1.4 Detección de esquinas

La detección de esquinas (*corner finding*) es una técnica usada en los sistemas de visión por computador con el fin de obtener ciertos rasgos característicos de una imagen. El objetivo de los detectores de esquinas es localizar puntos de la imagen en los que se crucen gradientes de intensidad en direcciones perpendiculares.

Una esquina puede definirse como la intersección de dos bordes. También puede definirse como un punto para el que hay dos direcciones de bordes dominantes y diferentes en una vecindad local del punto.

Un punto de interés es un punto en una imagen que tiene una posición bien definida y puede ser detectado de forma robusta. Esto significa que un punto de interés

puede ser una esquina pero también puede ser, por ejemplo, un punto aislado de intensidad local máxima o mínima, final de líneas, o un punto en una curva donde la curvatura es localmente máxima. En la práctica, los métodos de detección de esquinas son llamados detección de puntos de interés en general. [4] [42]

El primer intento de detección de esquinas lo propusieron Chris Harris y Mike Stephens en su *paper* "*A combined corner and edge detector*" en 1988 [43] a partir del cual se desarrolló el denominado Harris Corner Detector. Básicamente se trata de una función que encuentra la diferencia de intensidad para un desplazamiento en todas las direcciones, llegando al siguiente resultado:[44]

$$R = \det(M) - k(trace(M))^2$$

donde:

$$\det(M) = \lambda_1 \lambda_2$$

$$trace(M) = \lambda_1 + \lambda_2$$

λ_1 y λ_2 son los autovalores de la matriz de autocorrelación de la segunda derivada

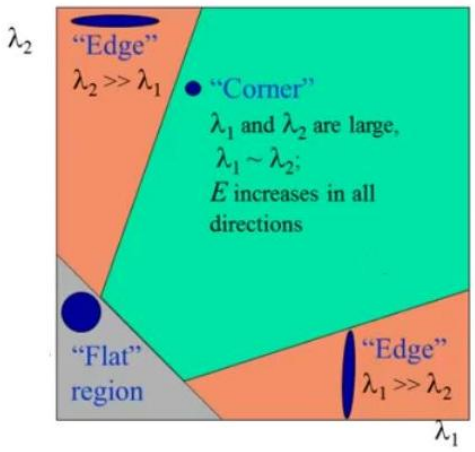

Figura 20: Representación de las esquinas de Harris

En 1994, J. Shi y C. Tomasi introdujeron ciertas mejoras en la función de Harris, en su *paper* "*GoodFeatures to Track*"[45], nombre que adquirió la función que desarrolló y que se aplica en este proyecto. Su propuesta es la siguiente:

$$R = \min(\lambda_1, \lambda_2)$$

Si es mayor que un valor umbral, se considera una esquina. Como se puede ver en la siguiente imagen, solo cuando ambos autovalores son mayores que el valor mínimo, puede considerarse una esquina.[46]

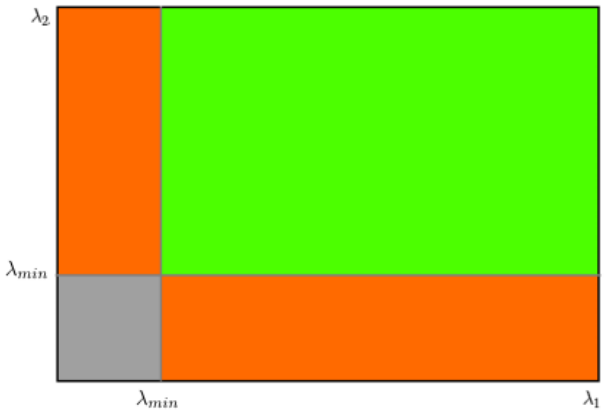

Figura 21: Representación de las esquinas de Shi-Tomasi

La función descrita es la siguiente: [4] [47]

```
goodFeaturesToTrack(InputArray image,OutputArray
corners,int maxCorners, double qualityLevel, double
minDistance,InputArray mask=noArray(), int
blockSize=3, bool useHarrisDetector=false, double
k=0.04);
```

donde el primer parámetro es la imagen que se pasa a la función y el segundo es el vector salida con las esquinas detectadas por la función. El resto son parámetros de calidad y ajuste, como el número de esquinas que se detectan o la distancia mínima entre ellas. En este caso la función queda:

```
goodFeaturesToTrack(grayOld,featuresOld,500,0.01,10
,Mat(),3,0,0.04);
```

4.1.5 Algoritmo Optical Flow

Optical Flow (flujo óptico) es el patrón del movimiento aparente de los objetos, superficies y bordes de una imagen causado por el movimiento relativo entre un observador y dicha imagen. Sus aplicaciones tales como la detección de movimiento, la

segmentación de objetos, el tiempo hasta la colisión y el enfoque de cálculo de expansiones, la codificación del movimiento compensado y la medición de la disparidad estereoscópica utilizan este movimiento de las superficies y bordes de los objetos. [4]

Los métodos de flujo óptico se basan en calcular el movimiento entre dos imágenes tomadas en tiempo t y t+Δt y asumen dos principios: la intensidad de los píxeles no cambia entre dos *frames* consecutivos y los píxeles vecinos deben presentar el mismo movimiento.[48] Son métodos diferenciales ya que se basan en aproximaciones de series de Taylor y usan derivadas parciales respecto a las coordenadas espaciales y temporales.

El método de Lucas-Kanade es un método diferencial ampliamente utilizado para la estimación de flujo óptico desarrollado por Bruce D. Lucas y Tadeo Kanade. Supone que el flujo es esencialmente constante en una zona local del píxel bajo consideración, y resuelve las ecuaciones básicas de flujo óptico para todos los píxeles en dicha zona, por el criterio de mínimos cuadrados. Al combinar información de varios píxeles cercanos, el método de Lucas-Kanade a menudo puede resolver la ambigüedad inherente de la ecuación de flujo óptico. También es menos sensible al ruido de la imagen que otros métodos. Por otra parte, al ser un método puramente local, no puede proporcionar información de flujo en el interior de las regiones uniformes de la imagen.

La función que se ha utilizado para controlar los movimientos de la imagen es la siguiente: [4]

```
void calcOpticalFlowPyrLK(InputArray
prevImg,InputArray nextImg, InputArray prevPts,
InputOutputArray nextPts, OutputArrays status,
OutputArray err);
```

donde el primer parámetro es el *frame* previo, el segundo es el *frame* actual (inmediatamente posterior al anterior), el tercero es el vector de esquinas obtenido en con el detector Shi-Tomasi en el paso anterior, el cuarto es un vector de entrada y salida con la identificación de los puntos anteriores. La función con nuestros parámetros queda de la siguiente forma:

```
calcOpticalFlowPyrLK(grayOld,
image_roi,featuresOld,featuresNew,featuresFound,err
);
```

4.1.6 Eliminación de outliers

Una vez se han obtenido los vectores con la posición de los nuevos puntos de interés, se necesita realizar un filtro que elimine los *outliers* que aparecen y que se confunden con los verdaderos movimientos que se producen entre *frames*. Para realizar ese filtro se calcula la distancia de todos los vectores obtenidos restando coordenada a coordenada las posiciones nuevas de los puntos (obtenidas del vector `featuresNew`) y las posiciones antiguas (`featuresOld`), y realizando la operación que proporciona su longitud.

Solo se aceptan aquellos movimientos cuya longitud se encuentre entre dos parámetros (`param_high` y `param_low`).

Así queda en nuestro programa:

```
x=(featuresNew[k].x)-(featuresOld[k].x);
y=(featuresNew[k].y)-(featuresOld[k].y);
distance=sqrt((x*x)+(y*y));
if((distance<param_high)&&(distance>param_low))
```

4.1.7 Filtro de los vectores de adelantamiento

Tras realizar el filtrado de *outliers* es el momento de determinar qué movimientos suponen realmente un posible adelantamiento, de entre todos los movimientos que se han producido entre *frames* consecutivos.

Al trabajar con vectores en 2D, tenemos cuatro posibles posiciones de vectores, una para cada cuadrante de un sistema x-y en el que se podría encuadrar dichos vectores.

Es importante realizar una aclaración previa cuyos efectos se verán posteriormente en la función que calcula el ángulo de cada vector: en *OpenCV*, el origen de coordenadas se encuentra situado en la esquina superior izquierda. Los valores crecientes de la coordenada x se encuentran a la derecha mientras que los valores

crecientes de la coordenada y se encuentran situados hacia abajo, tal y como se muestra en la siguiente figura. Esto implicaría una reordenación de los cuadrantes respecto a la visión común que tenemos de ellos.

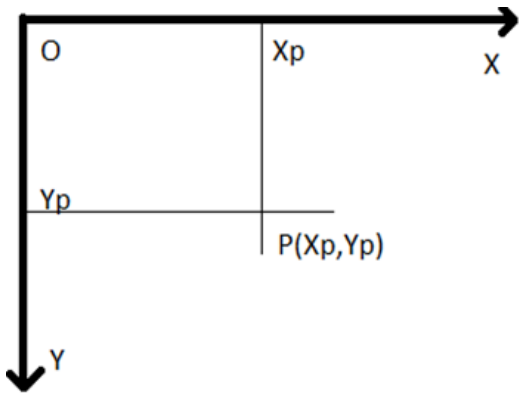

Figura 22: Dirección y sentido de los ejes en OpenCV

Para no confundir al lector de esta memoria, ni al usuario que lea el código, se ha optado por realizar una pequeña operación matemática en la función `atan2(y/x)` que permite reordenar los cuadrantes conformándolos de acuerdo a la visión habitual que se tiene de ellos, tal y como se puede ver en la siguiente imagen y en la función que se muestra a continuación. Por ello desde este momento todas las consideraciones de posiciones y coordenadas se realizarán de acuerdo a estos cuadrantes.

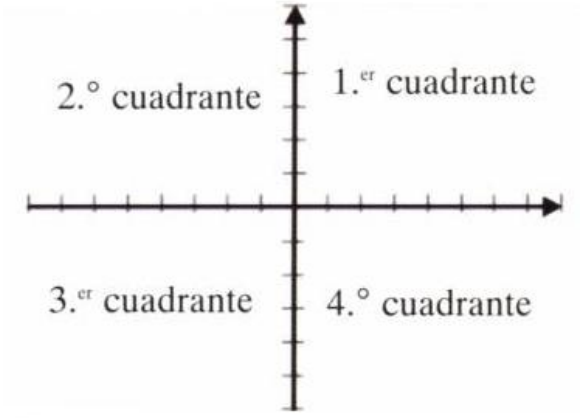

Figura 23: Dirección y sentido de los ejes cartesianos

Según el cuadrante en el que se encuentre el vector, este corresponderá a un tipo de movimiento respecto a la cámara embarcada:

– Primer cuadrante ($0º < \alpha < 90º$): Los vectores asociados a este cuadrante hacen referencia a vehículos que provocan el adelantamiento (en este

proyecto solo se tendrán en cuenta adelantamientos por el lado izquierdo del vehículo). En ellos tanto la coordenada x como la coordenada y de la posición del *frame* anterior son menores que en la posición del *frame* siguiente.

Figura 24: Vector de movimiento posicionado en el primer cuadrante

– Segundo cuadrante (90º < α < 180º): Los vectores con esta dirección y sentido serían aquellos provocados en dos tipos de circunstancias: adelantamientos por el lado derecho (como el que se puede observar en la figura 7), los cuales no se tendrán en cuenta; y aquellos producidos por objetos estáticos o con sentido contrario a la marcha del vehículo propio, que se encuentren sobre la línea de horizonte y la zona izquierda de la imagen.

Figura 25: Vector de movimiento posicionado en el segundo cuadrante

– Tercer cuadrante (180º < α < 270º): Los vectores con esta dirección y sentido son aquellos producidos por objetos estáticos o con sentido contrario a la marcha del vehículo propio, que se encuentren bajo la línea de horizonte y en la zona izquierda de la pantalla.

– Cuarto cuadrante (270º < α < 360º): Los vectores con esta dirección y sentido son aquellos producidos por objetos estáticos o con sentido contrario a la marcha del vehículo propio, que se encuentren bajo la línea de horizonte y en la zona derecha de la pantalla.

La función que calcula el ángulo dados dos puntos es la siguiente (se recuerda que se ha modificado para proporcionar los ángulos de acuerdo a los cuadrantes de la Figura 5):

```
static double angle = (atan2 (y,x) * 180.0 / PI)*(-1);
```

El algoritmo que diferencia los adelantamientos de acuerdo a la dirección de los vectores de movimiento es el siguiente:

```
if((angle<90)&&(angle>0))     //Adelantamiento
{
      line(image_roi,featuresOld[k],featuresNew[k],Scalar(0,0,255));
      featuresDetection[j]=featuresOld[k];
      j++;
      new_overtaking=true;
}
```

4.1.8 Cálculo de posición de adelantamiento

Una vez el algoritmo ha analizado todos los *features* detectados, eliminando los *outliers* y guardando los adelantamientos en el vector `featuresDetection`, se calcula la posición de del vehículo utilizando solo las detecciones producidas por adelantamientos. El código dedicado a realizar esta tarea es el siguiente:

```
for(int m=0; m<j; m++)
{
      static double average_x=+featuresDetection[m].x
      static double average_y=+featuresDetection[m].y
}

point.x = average_x/j;
point.y = average.y/j;
```

donde la variable `point` ha sido definida anteriormente como:

```
geometry_msgs::Point point;
```

4.1.9 Publicación de la posición de adelantamiento

Por último, el software está diseñado para publicar un topic que contiene las coordenadas del vehículo que provoca el adelantamiento.

Solo se publica cuando se encuentra un nuevo adelantamiento gracias a la variable *new_overtaking* que actúa como *flag* por lo que seguidamente se vuelve a poner a cero, a la espera de otra detección correcta:

```
ros::Publisher overtaking_pub = nh.advertise<geometry_msgs::Point>
("overtaking_xy",1);

ros::Rate loop_rate(10); // Hz

while(ros::ok()){
	ros::spinOnce();
	if(new_overtaking){
		overtaking_pub.publish(point);
		new_overtaking = 0;
	}
	loop_rate.sleep();
}
```

Capítulo 5

Análisis de resultados y conclusiones

"La conclusión es que sabemos muy poco y sin embargo es asombroso lo mucho que conocemos. Y más asombroso todavía que un conocimiento tan pequeño pueda dar tanto poder"

Bertrand Russell. Filósofo, matemático y escritor británico

5.1 PRUEBAS Y ANÁLISIS DE RESULTADOS

5.1.1 Ajustes para las pruebas

El proyecto fue realizado en un ordenador portátil con acceso a cámara web, con la cual se fueron realizando las primeras pruebas de captura de esquinas y seguimiento de puntos.

El nodo principal del proyecto (*cam_suscriber*) se suscribe al *topic usb_cam/image_raw* el cual transporta imágenes de dicha cámara.

Una vez el proyecto fue avanzando, encontramos la necesidad de realizar las pruebas necesarias tanto de captura como de seguimiento en una situación real, para lo cual se contó con un archivo *bag*, explicado en el capítulo 3, en el apartado relativo a *ROS*, que contenía imágenes tomadas desde una cámara embarcada en un vehículo.

A efectos de código, este cambio solo obligaba a modificar el *topic* al que se suscribía nuestro nodo principal, dejando en desuso además el nodo *usb_cam* que publicaba las imágenes de la cámara. Esta es una de las grandes virtudes de *ROS*, con apenas una pequeña variación podíamos suscribirnos a un *topic* diferente del que obtener imágenes distintas de las que trabajábamos hasta el momento, y a su vez distintas de las imágenes que se conseguirán con el sistema completo montado y operativo en el vehículo.

El cambio aparece reflejado a continuación:

```
image_transport::Subscriber sub = it.subscribe
("xb3_no_rect/left", 1, imageCallback);
```

Como se puede observar, el único cambio introducido respecto a la función definitiva que se vio en el capítulo 4, en su apartado correspondiente (Adquisición de imágenes) es el *topic* al que se suscribe el algoritmo.

A efectos prácticos, tras iniciar *ROS* con el comando `roscore` (capítulo 3), no se hacía correr el nodo *usb_cam*, sino que se accedía al contenido del *bag* en cuestión. Para ello se comunicaba al terminal el directorio donde se encontraba dicha bolsa y se introducía el comando necesario para reproducirlo, a saber:

```
cd ~/[carpeta_raíz]
rosbag play [archivo bag] -l
```

donde `-l` permite reproducir indefinidamente el contenido del archivo.

Una vez la reproducción se encuentra en marcha, accedemos a nuestro software mediante el comando `rosrun`, como se vio en el tercer capítulo.

5.1.2 Resultados de las pruebas

Una vez se tuvo el software funcionando en modo de prueba, se realizaron ajustes más finos en los siguientes aspectos:

- Reordenación del código: tras las primeras pruebas se observó que el proceso no era fluido, sino que necesitaba de demasiados ciclos para procesar cada *frame* que llegaba, lo que producía errores en el seguimiento de los puntos con la función *optical flow*. Las soluciones por las que se optaron fueron dos: en primer lugar se definió el ROI comentado en el capítulo anterior, que disminuía en un 85% (de acuerdo a los ajustes de tamaño del ROI) el tamaño de la imagen a analizar. En segundo lugar se optó por realizar una sola vez la detección de contornos (en la primera iteración) y trabajar en cada nuevo procesamiento de imagen con los vectores de puntos del *frame* anterior.

- Eliminación de *outliers*: se ajustaron los parámetros `param_high` y `param_low` de acuerdo al rango de longitudes que se considera permitido.

- Ajuste del tamaño del ROI: en el bag proporcionado, la cámara se encontraba apuntando a una zona relativamente baja, en cuya zona inferior podía verse gran cantidad de imagen del capó del vehículo, de ahí la elección de los parámetros `param_col` y `param_row` que situaban el ROI respecto a la imagen original en la esquina superior izquierda, como aparece en la figura 24.

5.2 CONCLUSIONES

En este proyecto se ha llevado a cabo el análisis de un aspecto relativo a la seguridad vial, desde el campo de la visión por computador, como es el aviso por adelantamientos.

Se ha apreciado como este conjunto de acciones y directrices relacionados con dicho campo son complejas, quizá no tanto desde el aspecto meramente teórico, gracias a herramientas tan versátiles como *OpenCV* y *ROS*, así como la comunidad de desarrolladores que las mejoran cada día, pero sí a la hora de conseguir resultados realmente óptimos.

La multitud de filtrados, conversiones, comparaciones y, en general, potentes funciones que se ven involucradas en un proyecto como este dan una idea del equipo hardware necesario no solo para su instalación final en el vehículo, sino también para el proceso de desarrollo y pruebas de estos últimos meses, principalmente en lo referente a la velocidad de procesamiento de las imágenes obtenidas a través de la cámara.

Una vez finalizado el proyecto, es posible hablar de las ventajas y limitaciones que han aparecido a lo largo del mismo. Las principales ventajas son las siguientes:

- Gracias al potente sistema computacional con el que cuenta el vehículo de pruebas de la Universidad Carlos III, el procesado de las imágenes será mucho más rápido y preciso, en comparación con el procesado obtenido usando simples ordenadores personales.

- Es capaz de detectar adelantamientos no solo en rectas, sino también en curvas y rotondas. De hecho, detecta cualquier adelantamiento dentro de la legalidad, entendiéndose esto último en términos de velocidad y por la izquierda. Como se verá a continuación, uno de los posibles trabajos futuros es detectar adelantamientos también desde el lado derecho del vehículo.

- Quizá la mayor ventaja del proyecto es la forma que tiene de acceder a las imágenes de la cámara proporcionada por la tecnología de mensajes y temas

de ROS, como se explicó en los capítulos tercero y cuarto, y como cambiar de fuente de imágenes, por ejemplo de un archivo *bag*, fácilmente.

También han aparecido ciertas limitaciones de las que debemos ser conscientes a la hora de realizar trabajos futuros sobre esta tecnología:

- Cada iteración que realiza el algoritmo en busca del seguimiento de los puntos con los que obtenemos los vectores de movimiento se volvía demasiado lenta, lo que produce errores aleatorios en el proceso. Aunque la mayoría de esos *outliers* han sido convenientemente eliminados, durante el periodo de pruebas queda patente que no todos son controlables.

- Debido al motivo anterior, adelantamientos excesivamente rápidos no podrían ser computables por un PC común.

- Por supuesto, este software requiere unas condiciones de iluminación mínimas. Como se comentará en el capítulo siguiente, una posible ampliación del proyecto sería realizar un estudio de situaciones reales de conducción con iluminación artificial (iluminación de la vía e iluminación propia de cada vehículo mediante faros).

Capítulo 6

Trabajos futuros

"El futuro tiene muchos nombres. Para los débiles es lo inalcanzable. Para los temerosos, lo desconocido. Para los valientes es la oportunidad"

Víctor Hugo. Novelista francés

Parte del trabajo ya está realizado pero es importante no conformarse y mirar adelante. En lo referente a esto, aparecen varias ideas como posibles ampliaciones o trabajos futuros derivados de este proyecto.

A continuación se exponen y explican superficialmente algunas de las ideas más interesantes que han surgido:

6.1 CÁLCULO DE LA VELOCIDAD DE ADELANTAMIENTO

Una vez se activa el *flag* de adelantamiento (`new_overtaking`) y se publica la posición del vehículo, es posible realizar un cálculo de la velocidad aproximada de adelantamiento, siempre teniendo en cuenta la velocidad propia; es decir, realmente se está calculando una velocidad relativa.

Se observará cuánto ha variado dicha posición, en cuánto tiempo ha ocurrido y se deberán extrapolar los resultados obtenidos (pixel/segundo) a una unidad métrica apropiada. Para ello se debe realizar un estudio y posteriormente una función que calcule la relación existente entre pixeles recorridos y distancia (en metros) recorrida en la realidad. Con ello y la velocidad del vehículo propio, proporcionada por el ordenador de a bordo, se puede estimar la velocidad absoluta con la que un vehículo está realizando el adelantamiento.

6.2 ADELANTAMIENTOS POR LA DERECHA

Aunque en carreteras tales como autovías y autopistas este tipo de adelantamientos están prohibidos, por todos es sabido que es práctica común entre algunos conductores. Inclusive, en poblado puede producirse un adelantamiento por la derecha dentro de la legalidad. Por ello es interesante poder detectar estos adelantamientos en caso de cambio de carril y así evitar una colisión.

Para ello habría que realizar ciertos ajustes:

- En primer lugar, se debería ajustar un ROI distinto al de nuestro proyecto, o mejor dicho, localizado en una zona distinta, concretamente en el lado derecho de la pantalla. En principio el tamaño podría ser el mismo e incluso sus parámetros de ajuste podrían permanecer invariantes. Bastaría con modificar el origen del rectángulo que da forma al ROI.
- En segundo lugar, se debería modificar, solo para este nuevo ROI en concreto, el ángulo que detecta como adelantamiento, ya que este no sería un vector del primer cuadrante, sino del segundo, como se aprecia en la figura 7 del capítulo 3.
- Sería también conveniente diferenciar a nivel de *flag* adelantamiento por la izquierda y por la derecha para evitar confusiones al conductor y al propio software. De esta forma podríamos pasar a tener: `new_overtaking_r` y `new_overtaking_l`.

6.3 CIRCULACIÓN NOCTURNA. ADAPTACIÓN A LUZ ARTIFICIAL Y/O ESCASA

Básicamente se puede asegurar que en condiciones de nula iluminación, o generalmente de visión deficiente, el software sería carente de utilidad, ya que su base es la visión.

A partir de esta consideración, se pueden realizar pruebas para saber cómo actuaría el software ante condiciones de luz artificial y la intensidad de la misma. Los ajuste necesarios serían casi con total seguridad muy abundantes y probablemente sería obligatorio una readaptación del algoritmo completo.

En cualquier caso sí se puede afirmar que en condiciones suficientes de luz artificial el algoritmo sigue funcionando perfectamente, como se ha podido comprobar en las pruebas realizadas durante los primeros pasos del desarrollo del software utilizando la cámara frontal del ordenador en cuartos iluminados exclusivamente con luz artificial.

6.4 ADAPTACIÓN AUTOMÁTICA DE PARÁMETROS SEGÚN LAS CONDICIONES LUMÍNICAS DEL EXTERIOR

Según el anterior apartado, y haciendo uso de sensores de medición de luz, como los que ya usan ciertos vehículos capaces de encender automáticamente sus faros cuando las condiciones de luz natural no son las óptimas.

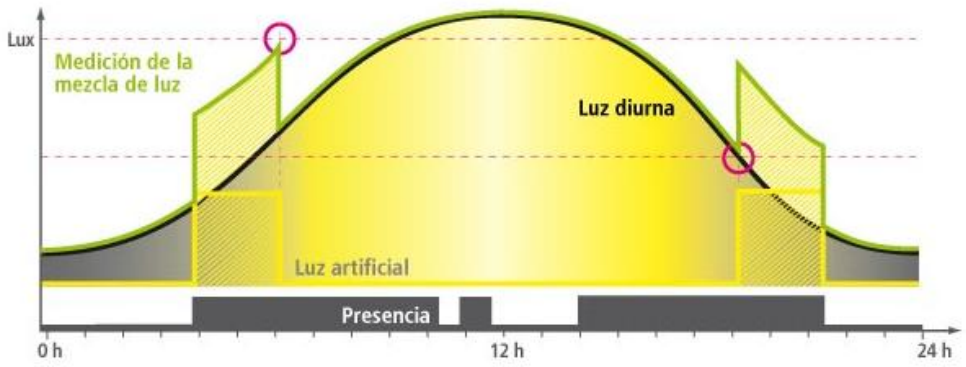

Figura 26: El detector de presencia mide la suma de luz artificial y luz diurna y conecta o desconecta la luz artificial según esa medición

Teniendo en cuenta esta información, y la obtenida durante las pruebas a distinta iluminación se podría realizar un algoritmo que tome como entrada el nivel de iluminación y ajuste los parámetros necesarios según las pruebas del trabajo anterior.

Capítulo 7

Bibliografía

"La información es poder"

Bill Gates. Empresario e informático estadounidense

[1] Seguridad vial http://pulitzercenter.org/projects/roads-kill-traffic-safety-world-health-organization-united-nations-fatalities-pulitzer-center-reporting-interactive-map-data-visualization(Agosto 2013).

[2] UAB Divulga, revista de investigación. Artículos disponibles en: www.uab.cat

[3] Sistemas de asistencia al conductor http://www.coches.net/noticias/sistemas-ayuda-conductor (Noviembre 2012)

[4] Bradski, G. & Kaehler, A., 2008. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media

[5] ROS http://www.ros.org/ (Agosto 2014)

[6] Vélez Serrano, J.F., Moreno Díaz, A.B., Sánchez Calle, Á., Esteban Sánchez, J.L., 2003. *Visión por computador*

[7] González Marcos, A. [et al.], 2006. *Técnicas y algoritmos básicos de visión artificial*. Servicio de Publicaciones de la Universidad de La Rioja

[8] Ruiz García, A. 2012. *Sistemas de Percepción y Visión por Computador*. Universidad de Murcia

[9] Berkeley Software Distribution http://www.bsd.org/ (Septiembre 2014)

[10] Grupo OpenCV de Yahoo https://groups.yahoo.com/neo/groups/opencv/info

[11] Grupo OpenCV de Stackoverflow http://stackoverflow.com/

[12] Conference on Computer Vision and Pattern Recognition http://www.informatik.uni-trier.de/~ley/db/conf/cvpr/cvpr2000.html (Septiembre 2014)

[13] Módulos de ROS http://docs.opencv.org/modules/core/doc/intro.html (Abril 2014)

[14] Brahmbhatt, S. 2013. *Practical OpenCV*. Apress

[15] Software Libre http://doc.ubuntu-es.org/Software_Libre (Junio 2012)

[16] Versión Ubuntu Precise https://translations.launchpad.net/ubuntu/precise (Enero 2014)

[17] Versiones de Ubuntu http://www.guia-ubuntu.com/index.php/Versiones_de_Ubuntu (Abril 2011)

[18] ROS http://wiki.ros.org/ (Diciembre 2013)

[19] Romero, A. & Fernández, E., 2013. *Learning ROS for Robotics Programming*. Packt Publishing

[20] Versiones ROS y Ubuntu http://wiki.gazebosim.org/wiki/DRC/ReleaseSchedule (Diciembre 2013)

[21] Distribuciones de ROS http://wiki.ros.org/Distributions (Julio 2014)

[22] Willow Garage http://www.willowgarage.com/pages/software/ros-platform (Junio 2014)

[23] ROS Packages http://wiki.ros.org/Packages (Marzo 2014)

[24] ROS Packages Tutorial http://wiki.ros.org/ROS/Tutorials/CreatingPackage (Mayo 2013)

[25] ROS Packages Tutorial http://wiki.ros.org/ROS/Tutorials/BuildingPackages (Diciembre 2012)

[26] ROS Nodes http://erlerobotics.gitbooks.io/erlerobot/es/ros/tutorials/understanding_ros_nodes.html

[27] ROS Nodes http://wiki.ros.org/Nodes (Febrero 2012)

[28] ROS Nodes Tutorial http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes (Marzo 2014)

[29] ROS Topics http://wiki.ros.org/Topics (Junio 2014)

[30] ROS Topics Tutorial http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics (Marzo 2014)

[31] ROS rostopics http://wiki.ros.org/rostopic (Enero 2014)

[32] ROS Publisher/Subscriber
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c%2B%2B) (Enero 2013)

[33] ROS Bags http://wiki.ros.org/Bags (Junio 2013)

[34] ROS rosbag comandos http://wiki.ros.org/rosbag/Commandline (Julio 2013)

[35] ROS roscore http://wiki.ros.org/roscore (Diciembre 2013)

[36] ROS usb_cam http://wiki.ros.org/usb_cam (Marzo 2014)

[37] OpenCV y ROS http://wiki.ros.org/opencv2 (Mayo 2014)

[38] Stack vision_opencv de ROS http://wiki.ros.org/vision_opencv (Marzo 2013)

[39] ROS image_geometry http://wiki.ros.org/image_geometry (Octubre 2010)

[40] ROS CvBridge
http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages (Junio 2014)

[41] Estrucutras básicas OpenCV
http://docs.opencv.org/modules/core/doc/basic_structures.html (Abril 2014)

[42] Features
http://docs.opencv.org/master/doc/py_tutorials/py_feature2d/py_features_meaning/py_features_meaning.html (Septiembre 2014)

[43] Detector de esquinas Harris, C. & Stephens, M., "A combined corner and edge detector" (1988). Paper disponible en:
http://courses.daiict.ac.in/pluginfile.php/13002/mod_resource/content/0/References/harris1988.pdf

[44] Harris features
http://docs.opencv.org/master/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html (Septiembre 2014)

[45] Shi, J. &Tomasi, C., "Good Features to Track" (1994). Paper disponible en: http://www.ri.cmu.edu/pub_files/pub2/shi_jianbo_1994_1/shi_jianbo_1994_1.pdf

[46] Shi-Tomasi
http://docs.opencv.org/master/doc/py_tutorials/py_feature2d/py_shi_tomasi/py_shi_tomasi.html (Septiembre 2014)

[47] Detección de esquinas http://docs.opencv.org/modules/imgproc/doc/feature_detection.html (Abril 2014)

[48] Lucas-Kanade http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html (Sept 2014)

[49] Instalación Ubuntu http://www.ubuntu.com/download/desktop/install-ubuntu-desktop (Abril 2014)

[50] Instalación Ubuntu http://www.wikihow.com/Install-Ubuntu-12.04 (Junio 2014)

[51] Instalación ROS Fuerte http://wiki.ros.org/fuerte/Installation/Ubuntu (Agosto 2012)

[52] Ubuntu y OpenCV https://help.ubuntu.com/community/OpenCV (Junio 2014)

[53] Instalación OpenCV en Ubuntu http://www.samontab.com/web/2012/06/installing-opencv-2-4-1-ubuntu-12-04-lts/ (Junio 2012)

[54] Instalación OpenCV en Ubuntu http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html (Abril 2014)

[55] Descargar OpenCV http://opencv.org/downloads.html (Septiembre 2012)

DESCARGA DE LINUX UBUNTU 12.04 LTS

Al comenzar el proyecto se optó por trabajar con la versión 12.04, en aquel momento la última versión LTS, que hace referencia a Long Term-Support, o soporte técnico extendido, y son aquellas que cuentan con actualizaciones de seguridad de paquetes software durante un periodo de 5 años (desde la aparición de 12.04, Precise Pangolin) tanto la versión de Escritorio como Servidor.

El primer paso es descargar el archivo ISO desde la página web de Ubuntu (http://www.ubuntu.com/download). Una imagen ISO es un archivo donde se almacena una copia o imagen exacta de un sistema de ficheros, en este caso el propio sistema operativo.

Desde el 17 de abril del 2104 se encuentra disponible la versión 14.04 LTS, Trusty Tahr.

INSTALACIÓN DE LINUX UBUNTU 12.04 LTS

Antes de la instalación es necesario realizar dos pasos previos: el primero consiste en copiar la imagen ISO en un CD/DVD; el segundo paso es configurar el ordenador para que arranque desde el propio CD/DVD.

Para ello, se iniciará la BIOS donde se debe cambiar el orden de preferencia o prioridad de arranque del ordenador. Se selecciona el CD/DVD como primera opción y se guardan los cambios realizados.

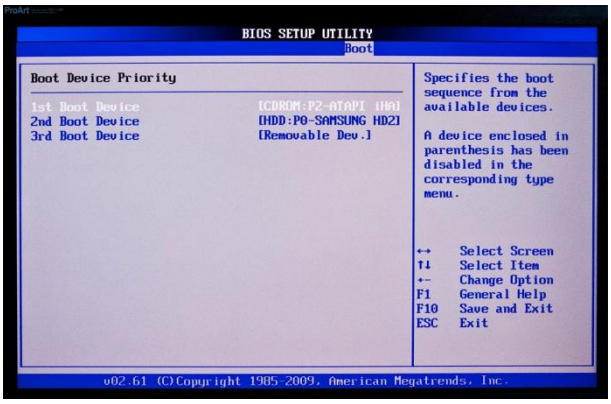

Figura 27: BIOS, selección de prioridad de dispositivo

A continuación comenzará la instalación de Ubuntu. Se trata de un proceso sencillo y guiado aunque merece la pena hacer ciertas aclaraciones: si se desea mantener un sistema operativo se deberá presionar la opción *Something else* de la pantalla *Installation type*, que permitirá al usuario decidir en qué partición de los discos duros se realiza la instalación.

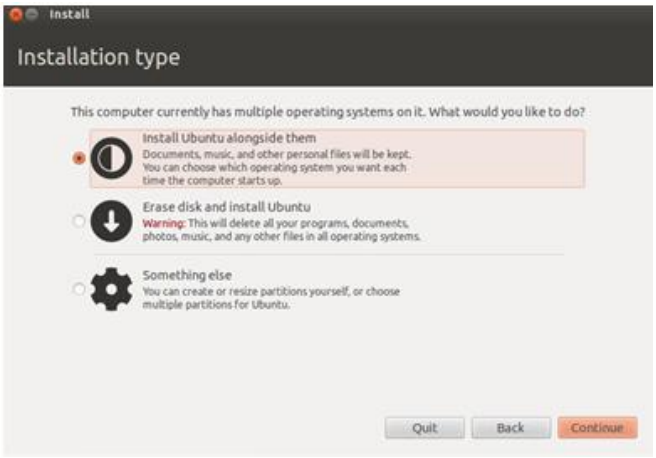

Figura 28: Menú de instalación de Ubuntu

Como paso previo a la instalación propiamente dicha, se pedirá al usuario datos como nombre, nombre de usuario y contraseña. El último paso es proceder a la instalación del sistema operativo. Una vez terminado, reiniciaremos el ordenador.

Hay que destacar que si existen múltiples sistemas operativos instalados, habrá que elegir cual deseamos abrir cada vez que encendamos o reiniciemos el ordenador.[49][50]

INSTALACIÓN DE ROS FUERTE SOBRE UBUNTU

La instalación de ROS Fuerte se realiza directamente desde el terminal de Ubuntu. Los pasos son los siguientes: [51]

– Configurar el ordenador para aceptar software desde los paquetes de ros.org. El comando a introducir varía según la versión de Ubuntu instalada. En nuestro caso:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
precise main" > /etc/apt/sources.list.d/ros-latest.list'
```

– Configurar las claves:

```
wget http://packages.ros.org/ros.key -O -| sudo apt-key add -
```

– A continuación se instala el sistema completo:

```
sudo apt-get install ros-fuerte-desktop-full
```

– Configuración de las variables de entorno: es un paso conveniente para que las variables de entorno de ROS sean añadidas automáticamente cada vez que se abre un nuevo terminal.

```
echo "source /opt/ros/fuerte/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

– Como paso final, se instalarán las utilidades python-rosinstall y python-resdep:

```
sudo apt-get install python-rosinstall python-rosdep
```

Si se desea comprobar la correcta instalación de ROS basta con introducir en el terminal el comando *roscore*, que como vimos anteriormente inicia ROS. Si aparece el siguiente mensaje la instalación habrá sido correcta:

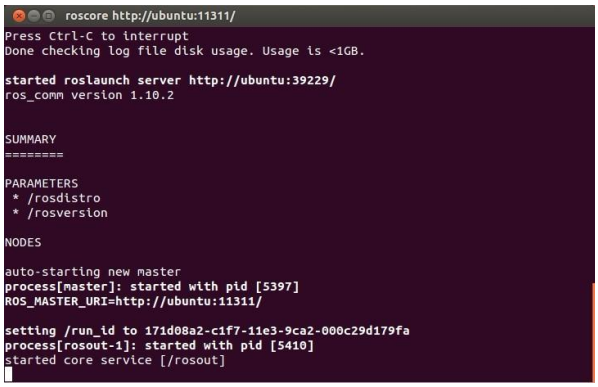

Figura 29: Activación del nodo roscore

DESCARGA E INSTALACIÓN DE OPENCV 2.4.2 PARA UBUNTU

La instalación de OpenCV 2.4.2 se realiza en su totalidad con el terminal. Los pasos a seguir son los siguientes: [52] [53] [54]

– Se actualiza el sistema:

```
sudo apt-get update
sudo apt-get upgrade
```

– Se instalan las dependencias de OpenCV:

```
sudo apt-get install build-essential libgtk2.0-dev libjpeg-
dev libtiff4-dev libjasper-devlibopenexr-devcmake python-dev
python-numpy python-tklibtbb-dev libeigen2-dev yasmlibfaac-
devlibopencore-amrnb-devlibopencore-amrwb-devlibtheora-
devlibvorbis-devlibxvidcore-dev libx264-dev libqt4-dev
libqt4-opengl-dev sphinx-common texlive-latex-extra libv4l-
dev libdc1394-22-dev libavcodec-devlibavformat-devlibswscale-
dev
```

– Se descarga y descomprime OpenCV. Puede realizarse desde la página web de OpenCV [55] o directamente desde el terminal introduciendo lo siguiente:

```
cd ~
wget
http://downloads.sourceforge.net/project/opencvlibrary/opencv
-unix/2.4.2/OpenCV-2.4.2.tar.bz2
tar -xvf OpenCV-2.4.2.tar.bz2
cd OpenCV-2.4.2
```

– Compilación e instalación. Desde el directorio de OpenCV creado, se introduce lo siguiente:

```
mkdir build
cd build
cmake -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON -D
WITH_V4L=ON -D INSTALL_C_EXAMPLES=ON -D
INSTALL_PYTHON_EXAMPLES=ON -D BUILD_EXAMPLES=ON -D WITH_QT=ON
-D WITH_OPENGL=ON ..
make
sudo make install
```

– A continuación se editará el archivo de configuración de OpenCV. Se abre con el siguiente comando, se añade al archivo /usr/local/lib, se guarda y se cierra el editor.

```
sudo gedit /etc/ld.so.conf.d/opencv.conf
```

– Se configura la librería introduciendo primero los siguiente comandos en el terminal:

```
sudo ldconfig
sudo gedit /etc/bash.bashrc
```

– Y se añaden las dos siguientes líneas al archivo que se acaba de abrir: PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/pkgconfig y export PKG_CONFIG_PATH. Tras guardar y salir del editor de texto, solo queda reiniciar el ordenador para finalizar la instalación.

NAVEGACIÓN A TRAVÉS DEL SISTEMA DE ARCHIVOS

Encontrar la dirección de un paquete	*rospack find [package_name]* *rosstack find [stack_name]*
Examinar las dependencias de 1er orden	*rospack depends1 [package_name]*
Cambiar el directorio de un paquete	*roscd [locationname[/subdir]]*
Listar el contenido de un directorio	*rosls [locationname[/subdir]]*

http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem

CREAR UN PAQUETE DE ROS

Crear un paquete	*roscreate-pkg [package_name] [dep1] [dep2] [dep3]*
Comprobar un paquete	*rospack profile* *rospack find [package_name]*
Las dependencias son guardadas en manifest.xml	*roscd [pachage_name]* *cat manifest.xml*

http://wiki.ros.org/ROS/Tutorials/CreatingPackage

CONSTRUIR UN PAQUETE DE ROS

Construir un paquete	*rosmake [package_name]*
Construir varios paquetes	*rosmake [package1] [package2] [package3]*

http://wiki.ros.org/ROS/Tutorials/BuildingPackages

NODOS DE ROS

Librerías cliente: rospy (Python), roscpp (C++)	
Nodo de activación de ROS	*roscore*

Listar nodos funcionando	*rosnode list*

Conseguir información de nodos	*rosnode info [node_name]*

Usar un nodo	*rosrun [package_name] [node_name]*

Cambiar el nombre de un nodo	*rosrun [package_name] [node_name] __name:=new_name*

Testar un nodo	*rsnode ping [node_name]*

http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes

TEMAS DE ROS

Usar rqt_graph (nodos y temas conectados)	*rosrun rqt_graph rqt_graph*
Usar rostopic	
help (opción de ayuda)	*rostopic –h*
echo (muestra los datos de un tema)	*rostopic echo [topic]*
	rostopic echo /turtle1/command_velocity
list (temas funcionando)	*rostopic list –h*
	rostopic list –v
type (devuelve tipo de mensaje de un tema)	*rostopic type [topic]*
	rosmsg show [massage_name]
pub (publica datos en un tema)	*rostopic pub [topic] [msg_type] [args]*
	rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 2.0 1.8
hz (reporta la frecuencia de publicación de los datos)	*rostopic hz [topic]*
Usar rqt_plot	*rosrun rqt_plot rqt_plot*

http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

SERVICIOS Y PARÁMETROS DE ROS

Usar rosservice	
list (servicios de los nodos funcionando)	*rosservice list*
type (devuelve el tipo de un servicio)	*roservice type [service_name]*
call	*rosservice call [service] [args]*
(sin argumentos)	*rosservice call clear*
(con argumentos)	*rosservice call spawn 2 2 0.2 "turtle2"*

Usar rosparam	
list (parámetros de nodos funcionando)	*rosparam list*
set (ajusta el valor de un parámetro)	*rosparam set [param_name] [value]*
(después de rosparam set, usar rosservice call clear)	
get (devuelve el valor de un parámetro)	*rosparam get [param_name]*
dump (escribir parámetros al archivo)	*rosparam dump [file_name]*
load (cargar archivos)	*rosparam [file_name] [namespace]*

http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

CREAR msg Y srv DE ROS

Crear archivos msg	*roscd [package_name]*
	mkdir msg
	echo "file_type name" > msg/Num.msg
	# rosbuild_genmsg() (uncomment in CMakeList.txt)

Use rosmsg	
show (mostrar msg)	*rosmsg show [message_type]*
	rosmsg show beginner_tutorials/Num
help (opción de ayuda)	*rosmsg –h*
	rosmsg show -h

Crear archivos srv	*roscd [package_name]*
	mkdir srv
	roscp [package_name] [file_to_copy] [copy_path]
	roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv

	# rosbuild_gensrv() (uncomment in CMakeList.txt)

Usar rossrv	
show (mostrar srv)	*rossrv show [service_type]*
	rossrv show beginner_tutorials/AddTwoInts

Tras crear un nuevo msg o srv	*rosmake [package_name]*

http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv

ESCRIBIR UN NODO PUBLICADOR Y SUSCRIPTOR (C++)

Nodo Publisher	
Crear un archivo	src/pubisher_node_name.cpp src/talker.cpp
Código del publisher	

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

/**
 * This tutorial demonstrates simple sending of messages
over the ROS system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv
so that it can perform
   * any ROS arguments and name remapping that were
provided at the command line. For programmatic
   * remappings you can use a different version of
init() which takes remappings
   * directly, but for most command-line programs,
passing argc and argv is the easiest
   * way to do it.  The third argument to init() is the
name of the node.
   *
   * You must call one of the versions of ros::init()
before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "talker");

  /**
   * NodeHandle is the main access point to
communications with the ROS system.
   * The first NodeHandle constructed will fully
initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
```

```
   * The advertise() function is how you tell ROS that
you want to
   * publish on a given topic name. This invokes a call
to the ROS
   * master node, which keeps a registry of who is
publishing and who
   * is subscribing. After this advertise() call is
made, the master
   * node will notify anyone who is trying to subscribe
to this topic name,
   * and they will in turn negotiate a peer-to-peer
connection with this
   * node.  advertise() returns a Publisher object which
allows you to
   * publish messages on that topic through a call to
publish().  Once
   * all copies of the returned Publisher object are
destroyed, the topic
   * will be automatically unadvertised.
   *
   * The second parameter to advertise() is the size of
the message queue
   * used for publishing messages.  If messages are
published more quickly
   * than we can send them, the number here specifies
how many messages to
   * buffer up before throwing some away.
   */
  ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);

  /**
   * A count of how many messages we have sent. This is
used to create
   * a unique string for each message.
   */
  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data,
and then publish it.
     */
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    /**
     * The publish() function is how you send messages.
The parameter
     * is the message object. The type of this object
must agree with the type
     * given as a template parameter to the
advertise<>() call, as was done
     * in the constructor above.
     */
    chatter_pub.publish(msg);
```

```
    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }


  return 0;
}
```

Nodo Subscriber	
Crear un archivo	src/subscriber_node_name.cpp src/listener.cpp
Código subscriber	

```
#include "ros/ros.h"
#include "std_msgs/String.h"

/**
 * This tutorial demonstrates simple receipt of messages
over the ROS system.
 */
void chatterCallback(const std_msgs::String::ConstPtr&
msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv
so that it can perform
   * any ROS arguments and name remapping that were
provided at the command line. For programmatic
   * remappings you can use a different version of
init() which takes remappings
   * directly, but for most command-line programs,
passing argc and argv is the easiest
   * way to do it.  The third argument to init() is the
name of the node.
   *
   * You must call one of the versions of ros::init()
before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "listener");

  /**
   * NodeHandle is the main access point to
communications with the ROS system.
   * The first NodeHandle constructed will fully
initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The subscribe() call is how you tell ROS that you
want to receive messages
```

```
   * on a given topic.  This invokes a call to the ROS
   * master node, which keeps a registry of who is
publishing and who
   * is subscribing.  Messages are passed to a callback
function, here
   * called chatterCallback.  subscribe() returns a
Subscriber object that you
   * must hold on to until you want to unsubscribe.
When all copies of the Subscriber
   * object go out of scope, this callback will
automatically be unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is
the size of the message
   * queue.  If messages are arriving faster than they
are being processed, this
   * is the number of messages that will be buffered up
before beginning to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000,
chatterCallback);

  /**
   * ros::spin() will enter a loop, pumping callbacks.
With this version, all
   * callbacks will be called from within this thread
(the main one).  ros::spin()
   * will exit when Ctrl-C is pressed, or the node is
shutdown by the master.
   */
  ros::spin();

  return 0;
}
```

Construcción de nodos	
Añadir al final de CMakeLists.txt	*rosbuild_add_executable (talker src/talker.cpp)*
	rosbuild_add_executable (listener src/listener.cpp)
Construir el paquete	*rosmake [package_name]*

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

ESCRIBIR NODO SEEVICIO Y CLIENTE(C++)

Nodo Service	
Crear un archivo	src/service_node_name.cpp src/add_two_ints_server.cpp

Código del Service	`#include "ros/ros.h"`
	`#include "beginner_tutorials/AddTwoInts.h"`
	
	`bool add(beginner_tutorials::AddTwoInts::Request &req,`
	`beginner_tutorials::AddTwoInts::Response &res)`
	`{`
	`res.sum = req.a + req.b;`
	`ROS_INFO("request: x=%ld, y=%ld", (long int)req.a,`
	`(long int)req.b);`
	`ROS_INFO("sending back response: [%ld]", (long`
	`int)res.sum);`
	`return true;`
	`}`
	
	`int main(int argc, char **argv)`
	`{`
	`ros::init(argc, argv, "add_two_ints_server");`
	`ros::NodeHandle n;`
	
	`ros::ServiceServer service =`
	`n.advertiseService("add_two_ints", add);`
	`ROS_INFO("Ready to add two ints.");`
	`ros::spin();`
	
	`return 0;`
	`}`

Nodo Client	
Crear un archivo	src/client_node_name.cpp src/add_two_ints_client.cpp

Código Client	`#include "ros/ros.h"`
	`#include "beginner_tutorials/AddTwoInts.h"`
	
	`bool add(beginner_tutorials::AddTwoInts::Request &req,`
	`beginner_tutorials::AddTwoInts::Response &res)`
	`{`
	`res.sum = req.a + req.b;`
	`ROS_INFO("request: x=%ld, y=%ld", (long int)req.a,`
	`(long int)req.b);`
	`ROS_INFO("sending back response: [%ld]", (long`
	`int)res.sum);`
	`return true;`
	`}`
	
	`int main(int argc, char **argv)`
	`{`
	`ros::init(argc, argv, "add_two_ints_server");`
	`ros::NodeHandle n;`

```
  ros::ServiceServer service =
n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

Construcción de nodos	
Añadir al final de CMakeLists.txt	*rosbuild_add_executable (add_two_ints_server src/add_two_ints_server.cpp)*
	rosbuild_add_executable (add_two_ints_listener src/ add_two_ints_listener.cpp)
Construir el paquete	*rosmake [package_name]*

http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29

USO DE LOS DATOS DE ROS

Guardar temas publicados	*mkdir ~/bagfiles*
	cd ~/bagfiles
	rosbag record -a
Examinar archives de datos	*rosbag info <your_bagfile>*
Reproducir datos	*rosbag play <your_bagfile>*
Guardar un subconjunto de datos	*rosbag record –O subset [topic] [datatype]*
	rosbag record –O subset /turtle1/command_velocity /turtle1/pose

http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data